Distributed Systems Engineering
Submission Document
SUPD (Status Update)

Each team member must enter his/her personal data below:

Team member 1	
Name:	Lukas Greiner
Student ID:	11807161
E-mail address:	a11807161@unet.univie.ac.at

Team member 2	
Name:	Moritz Renkin
Student ID:	11807211
E-mail address:	a11807211@unet.univie.ac.at

Team member 3	
Name:	Isabel Pribyl
Student ID:	11807182
E-mail address:	a11807182@unet.univie.ac.at

Team member 4	
Name:	Nikita Glebov
Student ID:	01468381
E-mail address:	nikegleb@yandex.ru

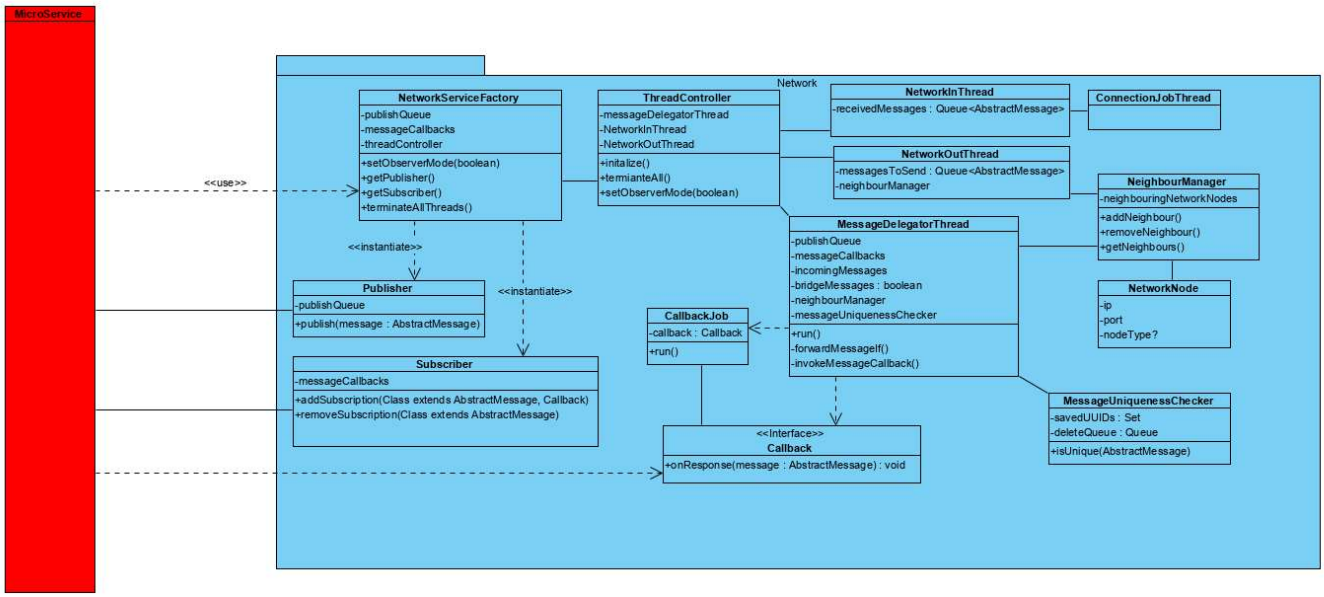
Messaging Framework Specification (MF):

Design Decisions:

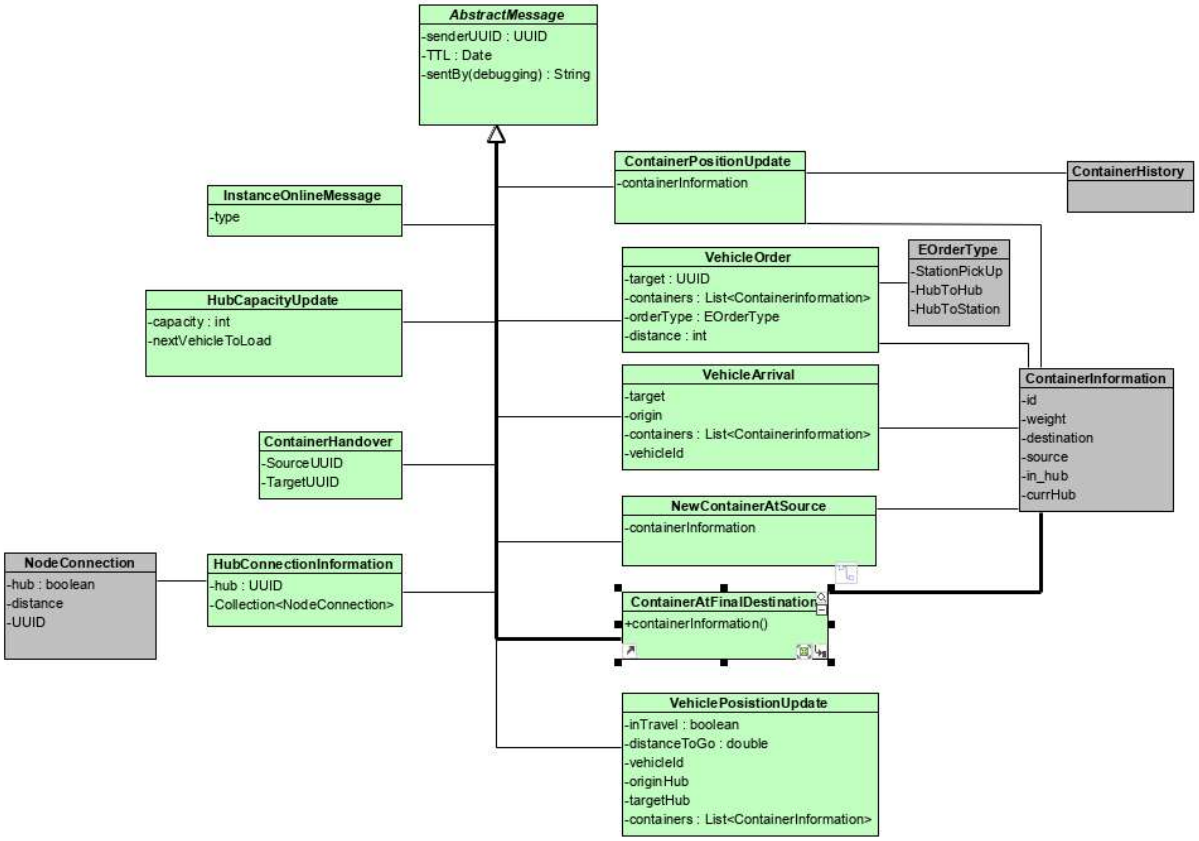
In order to fulfil assignment requirements, network of microservices (MS) should be created. Ability to transfer messages must be implemented in message framework. MS is divided into several entities: Hub, Hub Operator, Container, Vehicle, Source/Destination. Each entity can be „bridged “with other MSs, which means that two MS are connected and can send their messages directly to other, for them to acquire and forward it to another member of the network. To do this, we represented each MS as a „Node “, each “Node” has its own neighbours (bridged Nodes”). Nodes can only communicate directly only with their neighbours (exception: Hub Operator, that can receive all the messages, that it needs). Therefore, if Node wants to communicate with other Nodes from the network, that are not connected to, they should send it to all their neighbours for them to forward it to destination. Main concern was, how to identify if MS needs message, that flows through the network. For that case combination of Subscribe functionality and Callback technology were used. „Subscribe “allows MS to obtain only certain type of messages and callback were used to save and handle only those messages, that are relevant for this type of MS. As for transport protocol, we use TCP, as it is reliable and does not require a lot of effort to implement.

UML Class Diagram(s)

Message Framework:



Messages:



Integration:
For simple usage of subscribe feature we separated message into different types, such as: HubCapacityUpdate, VehicleArrival, NewContainerAtSource and so on. Using this division of messages, we have implemented subscription only to certain types of messages, so that MS receiver will not be overflown by number of messages. Because of the concurrent operations, that are happening during whole usage of a program, we implemented a “callback” feature, that is very handy in concurrent field. That allows us to filter messages without interrupting any processes. In that case, MS job is to specify type of message and to tune callback mechanics on its side. Callbacks are always executed automatically by the MS when the subscribed Message arrives. As for library, we use .jar file to implement our MF in functionality of MS.

As for publish side, MF allows MS to send every message, that is included in MF. In order to do that, MS should use one of the formats, that is included in AbstractMessage.

Messages & Communication:
The MF only listens to the messages NewContainerAtSource and ContainerHandover which is sent by the containers to dynamically change the bridging neighbours when Containers are moved around.

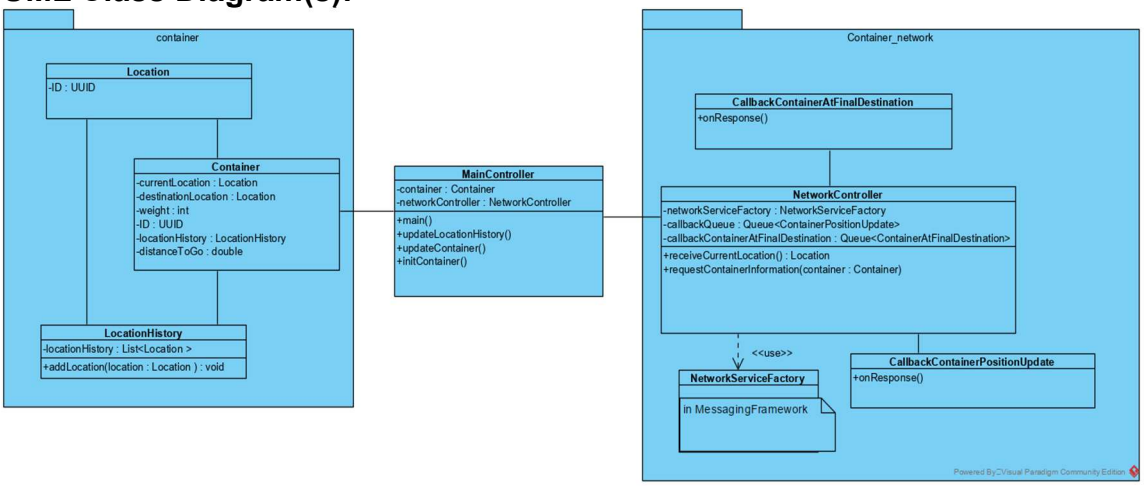
Loop prevention is ensured by a UUID that is carried by each message and then saved by the Message Framework. If the same message is received again, it is dumped, neither bridged nor processed.

Status:
The interfaces for subscribing and publishing Messages is complete. The callbacks execution at message arrival is also implemented already but not yet unit tested. The thread management, including the usage of thread-safe data structures is already adopted.
The dynamic adding and removal of neighbours when Containers spawn/move are not implemented yet as well as low level nework access (socket config etc.) and the management of IPs ports of the individual neighbours.

Microservice Containers [Smart Tags]:

Design Decisions:
Container MS is divided into two packages: Container and network. Container is responsible for Container itself, contains information like current Location, destination, weight, id, location history and distance to next Hub. network is package about connection between MS Container and MF. It is a place, where callbacks are being triggered. Other MS can receive current container location as well as receive container information.

UML Class Diagram(s):



Deployment:
The container is started with the following start parameters:

- UUID
- Weight
- Source
- Destination
- Port

Messages & Communication:
(For further information see Documentation of the Messages & Communication at the end of the document.)

Listener of:

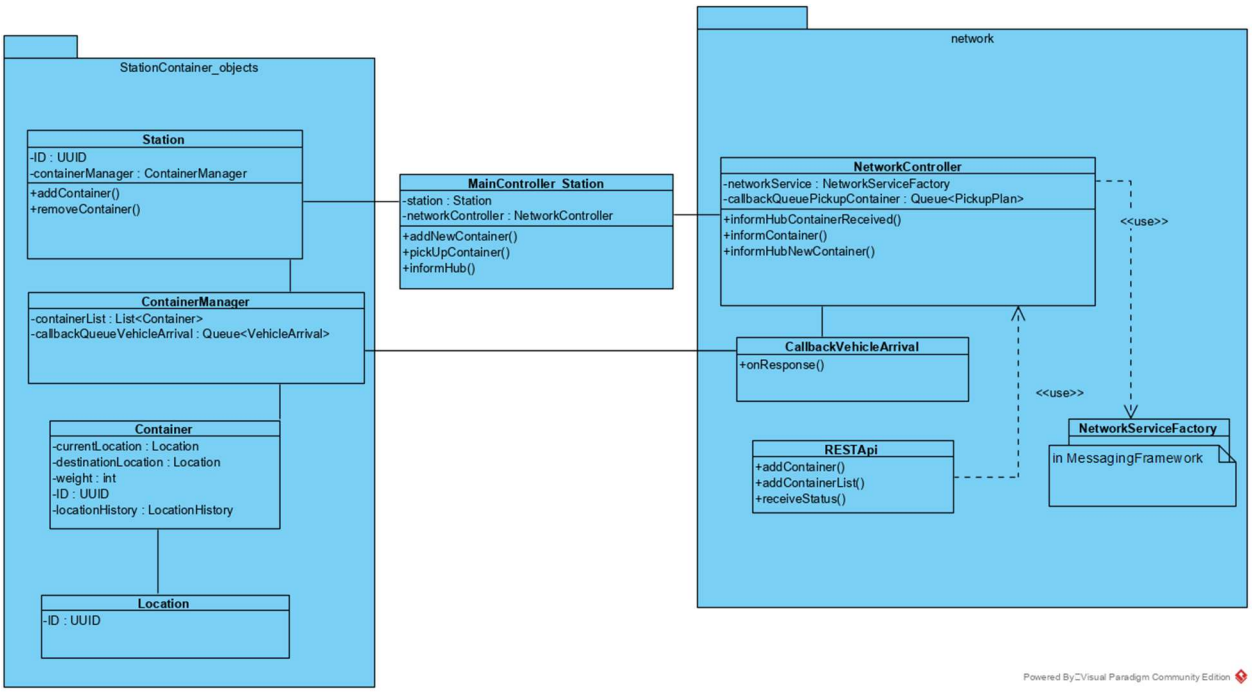
- VehiclePositionUpdate: Container needs to monitor its current position for its history
- ContainerAtFinalDestination: if they receive that the container is delivered it adds the location of the destination and then terminates itself
- ContainerPositionUpdates: The Container has to update itself with this information

Status:
Container package is close to being finished, structure of NetworkConntroller, MainController is presented as well as base for callbacks implementation.
Lack of internals of functions, changes in the number and type of messages are possible.

Microservice Container Sources/Destinations:

Design Decisions:
The Container “Sources/Destinations” has been renamed to “Station” for simplicity reasons. The Station is controlled by the MainController which is responsible for updating and calling the other components. The whole station is also divided into the objects and the network package. The objects package provides the logic for collecting, managing and providing container objects. These are than distributed by the NetworkController. It is also responsible to call the other microservices if a necessary action is needed. All incoming traffic is managed by subscribing to the VehicleArrival message which contain all the information need.

UML Class Diagram(s):



Deployment:
The station microservice gets deployed to one or multiple of our machines and is than started manually. For a successful start-up a separate .config file is provided for each instance. These files include parameters like the port number, the direct neighbours of an instance and the hub operator. The starting mechanism is simply executing the compiled .jar file on one of our machines with a path for the startup.config file.

Starting command:
java -jar ms_station.jar “/path/to/properties/file/**startup.config**” <UUID>

Parameters in startup.config file:
port: 6502 (Increases by each instance)
neighbours: [http://localhost:6501,...]
hubOperator: http://localhost:6500

Web UI Rest API:

Get all containers in source node:

Title	Get all containers in source node
Description and Use Case	Get a list of all containers which have been injected to the source and not yet picked up. This can be used to see if containers are being picked up.
Transport Protocol Details	REST, endpoint: GET /source
Sent Body Example	Empty Request Body
Sent Body Description	---
Response Body Example	
Response Body Description	• containers: Array of containers
Error Cases	• Internal Server Error (should never happen only for debugging)

Insert single Container into the Source:

Title	Insert single Container into the Source
Description and Use Case	This operation inserts a single container into the Source. It will then be transported through the transport network to the specified destination.
Transport Protocol Details	REST, endpoint: POST /source/insert
Sent Body Example	
Sent Body Description	UUID: A unique identifier that will be used to identify the container along the way and at the pickup. Destination: UUID of the desired Destination Weight: Weight of the container in kg
Response Body Example	Empty response body
Response Body Description	---
Error Cases	• Destination node was not found: 404

Insert bulk of container:

Title	Insert bulk of container

Description and Use Case	This Operations inserts a specified number of containers at the source node in order to assess the behaviour of the transport network.
Transport Protocol Details	REST, endpoint: POST /source/insert/bulk

Sent Body Example	{
	"containers": [
	{
	"UUID": "string",
	"destination": "string",
	"weight": 20
	}
	]
	}

Sent Body Description	UUID: A unique identifier that will be used to identify the container along the way and at the pickup. Destination: UUID of the desired Destination Weight: Weight of the container in kg
Response Body Example	Empty response body
Error Cases	• Destination node was not found: 404

Get containers in destination node:

Title	Get containers in destination node
Description and Use Case	Get all containers which have been received at the destination node and are ready to be picked up (deleted).
Transport Protocol Details	REST, endpoint: GET /destination
Sent Body Example	Empty request body
Sent Body Description	---

Response Body Example	{
	"destination": "string",
	"containers": [
	{
	"UUID": "string",
	"source": "string",
	"destination": "string",
	"weight": 20
	}
	]
	}

Response Body Description	UUID: The unique identifier of the container. Source: Source of the container Destination: UUID of the Destination (should always be equal to UUID of contacted destination) Weight: Weight of the container in kg
Error Cases	• Internal Server Error (should never happen only for debugging)

Delete a container:

Title	Delete a container

Description and Use Case	Pickes up (deletes) a container from the destination node and returns the container properties. Container must be in the destination. Note, source and destination are logically seperated in this regard.
Transport Protocol Details	REST, endpoint: DELETE /destination/pickup/{containerId}
Sent Body Example	Empty request body
Sent Body Description	---
Response Body Example	
Response Body Description	UUID: The unique identifier of the container Source: Source of the container Destination: UUID of the Destination (should always be equal to UUID of contacted destination) Weight: Weight of the container in kg
Error Cases	• Container not found: 404 • Internal Server Error (should never happen only for debugging)

Messages & Communication:
(For further information see Documentation of the Messages & Communication at the end of the document.)

Publisher of:
- ContainerAtFinalDestination: sends this message if a container is delievered to it and it is the final destination of the container.
- NewContainerAtSource: sends out this message if a new container gets injected to a Station
- InstanceOnlineMessage: sends this message out when online
- ContainerHandover: needs to be sent when a container is handed over to a new Vehicle for the MF to configure the new bridging connection.

Listener of:
- VehicleArrival: if a vehicle arrives at a station it checks if the vehicle contains container that have their station as a destination and if that is the case if knows the containers have arrived at its destination.
- StartSignal: MS can now start its service.

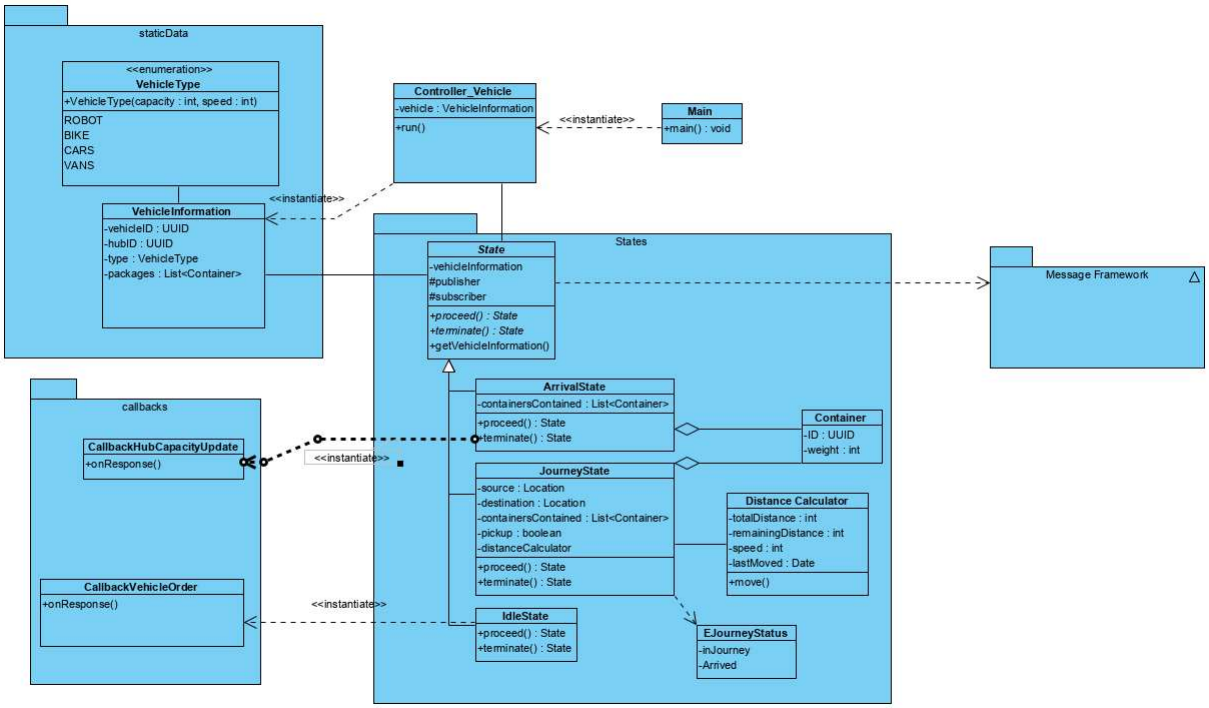
Status:
The basic Structure is implemented and the spring-boot web starter and the spring-boot gradle plugin were added as dependencies. The concrete implementation is still to implement.

Microservice Vehicles:

Design Decisions:

A state-based design was chosen for the Vehicle Microservice. The three different states that a vehicle can be in are: Journey, Idle and Arrival. They are all represented as a class that inherits from AbstractState which has the two methods proceed and terminate. Both of these return a new (or the same) state. This design is really useful for this Microservice because vehicles go through these different states repeatedly and have very different responsibilities during each one. For example, a Vehicle will not do anything but wait for a VehicleOrder in IdleState while it has to simulate a real vehicle by calculating the remaining distance and publish LocationUpdates etc when in JourneyState.
Another advantage of this design is the great abstraction of each state which makes them easily testable.

UML Class Diagram(s):



Deployment:

The vehicle instance must be passed the UUID of the hub which is it assigned to as starting parameter. The instance will then pass the information that it is online through the InstanceOnlineMessage, which will be processed by the Hub.
For this to work it is important that vehicles are only started when all Hubs are already running.

Messages & Communication:

(For further information see Documentation of the Messages & Communication at the end of the document.)

Publisher of:

- VehiclePositionUpdate: sends out an update of its position information and containers.
- VehicleArrival: if the vehicle arrives at a hub it sends the information is here and is ready to off load

- InstanceOnlineMessage: sends this message out when online
- ContainerHandover: needs to be sent when a container is handed over to a new Hub/Station for the MF to configure the new bridging connection.

Listener of:

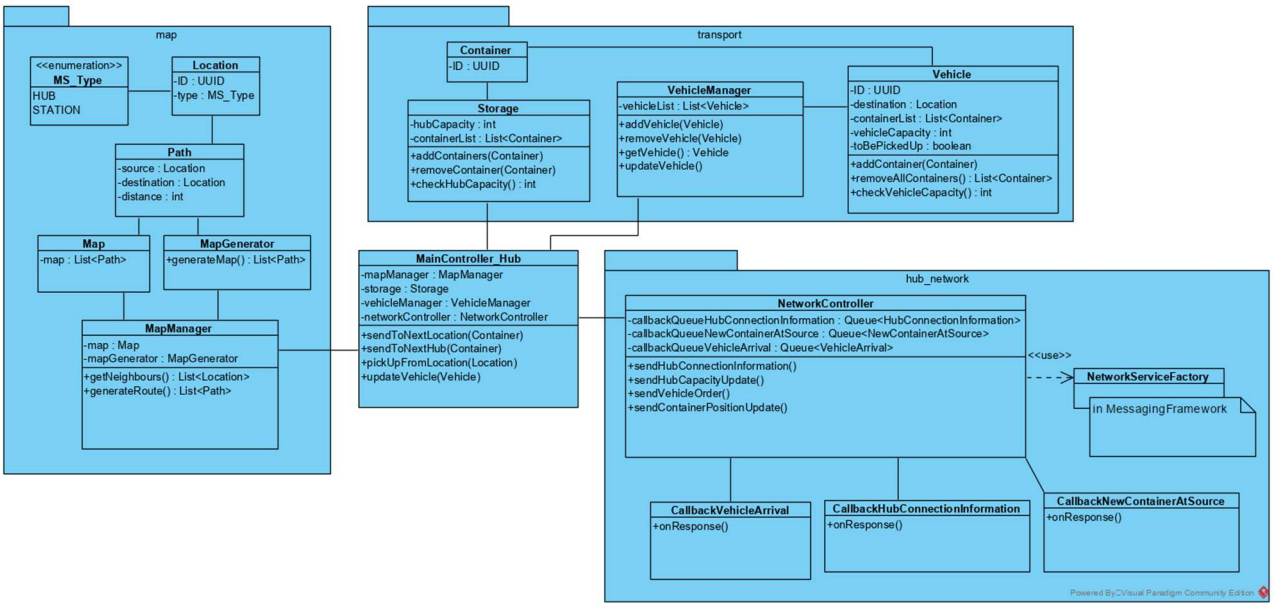
- HubCapacityUpdate: if the vehicle is waiting at the hub it checks if it is the next vehicle to load, otherwise it waits at hub
- VehicleOrder: uses this message for its state. If a new order arrives it has to transport the containers to the target or pickup containers and uses the given distance to simulate its journey.
- StartSignal: MS can now start its service.

Status:
The classes which store static information about the vehicle, containers etc. are finished. The states are semi-implemented with the core of the proceed logic done but no terminate logic in place. The publishing and subscribing of the states is not yet implemented. Skeleton for all other classes is done.

Microservice Hubs:

Design Decisions:
The Hub uses the NetworkController to store Queues of the different messages of the used callback classes. The callback classes in the Hub (CallbackVehicleArrival, CallbackHubConnectionInformation, CallbackNewContainerAtSource) use the onRepsonse method to add incoming messages to the Queues in the Network Controller. Provided by the MF, the Hub uses Subscriptions of the needed Messages. The NetworkController also is responsible for sending out messages with the use of Publisher provided by the MF.

UML Class Diagram(s):



The Hub contains three packages, map, transport and hub_network. The map package contains the structure of the Map, which works with Paths from one to another destination and they distance. The Hub uses this calculate the routing for the containers, which is all done in the MapManager class. The package transport handles the hub storage for containers, as well as the managing of assigned Vehicles. The hub_network package handles the communication (sending out messages and callbacks of incoming messages). The MainController class serves as a connection of all the aspects (map managing, vehicle managing, storage monitoring and network communication).

Deployment:
When starting the Hub, it will be given its UUID as a start parameter. The port will be assigned by the config file. From the config file, the hub also reads out its neighbours which it is connected to, as well as the UUID of the Hub Operator. After that the Hub sends out the InstanceOnlineMessage to signalise the hub is online and ready to listen to messages.

Messages & Communication:
(For further information see Documentation of the Messages & Communication at the end of the document.)

Publisher of:

- HubConnectionInformation: sends out its connection Information of connected Stations and Hubs
- HubCapacityUpdate: this message serves two purposes:
    - if a waiting vehicle can off load all of its containers, hub sends this message so the vehicle knows it can off load.
    - The message is also used to identify the hub capacity. Therefore, every time a vehicle offloads, the hub needs to send out the update of the free capacity.
- VehicleOrder: sends out which containers need to be picked up or transported to a certain target
- ContainerPositionUpdates: if a new container arrives and gets in its storage, it sends out the new position of the container
- InstanceOnlineMessage: sends this message out when online
- ContainerHandover: needs to be sent when a container is handed over to a vehicle Station for the MF to configure the new bridging connection.

Listener of:

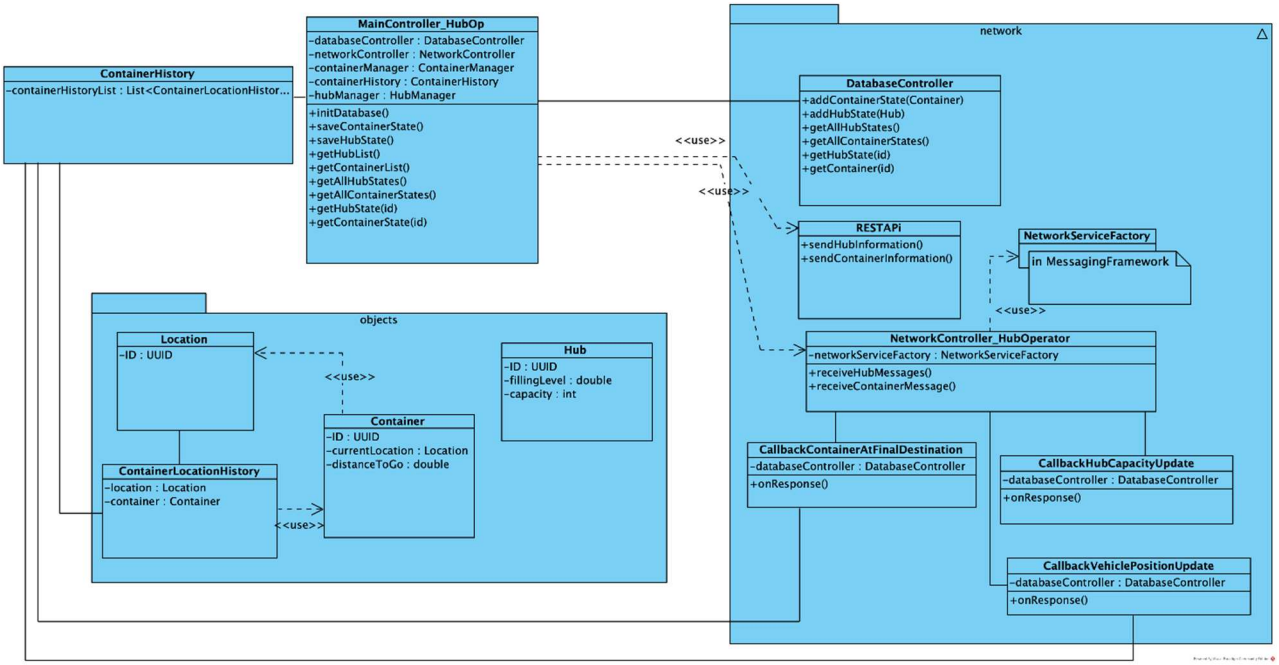
- HubConnectionInformation: receives the connection information of other hubs to generate the whole map for routing purposes
- NewContainerAtSource: needs to listen to this message because they maybe need to pick up the container if they are connected to the Station
- VehicleArrival: if the vehicle arrives at the hub, the hub needs to check its capacity and determine if it has enough space. Also, the hub needs to update the container if the package has arrived
- StartSignal: MS can now start its service.

Status:
So far, the structure of the hub, as well as a skeleton has been achieved. What's missing is the implementation of the classes. Also missing is the handling of the messages StartSignal.

Microservice Huboperator:

Design Decisions:
The Huboperator is divided into two parts: the networking part and the object part. Both parts are controlled by the MainController which is responsible for all basic actions and in which order they are taken. For the networking part we are using callbacks to evaluate all incoming responses and save part of the provided information in our database via the DatabaseController. Here comes the object part into play, all incoming responses are mapped to these objects. These objects are then used by the hibernate framework to easily store the data into the database. For the database we use a mysql instance as it is easily available and easy to install. The WebUI is provided with data via our REST API, which simply reads all the data saved in our database and upon a request and sends back a simple JSON file.

UML Class Diagram(s):



Deployment:
The hub operator will be only deployed on one of our machines and only one instance of it will be manually started. For configurating the hub operator on start-up, only basic information (like the Port number) is provided. The starting mechanism is simply executing the compiled .jar file on one of our machines with a path for the startup.config file. This file contains all the basic information necessary for completing the start-up.

Starting command:
java -jar ms_huboperator.jar “/path/to/properties/file/**startup.config**”

Parameters in startup.config file:

Port number: 6500

The hub operator is reachable by a REST API under the following URLs:

http://localhost:8080/hubOccupation
http://localhost:8080/containerLocation
http://localhost:8080/locationHistory

Web UI Rest API:
Get a list of all hubs and their current occupation

Title	Get a list of all hubs and their occupation
Description and Use Case	Get a list of all hubs and their occupation. This will be used to track the current status of all hubs available
Transport Protocol Details	REST, endpoint: GET /hubOccupation
Sent Body Example	-
Sent Body Description	-
Response Body Example	`[ { "hub": "c7c7482d-da86-44ec-ad35-713236251319", "occupation": 0.2 } ]`
Response Body Description	• hub: Id of requested hubs • occupation: filling level of requested hub in percent
Error Cases	• A hub-operator node was not found: 404 • Internal Server Error (should never happen only for debugging)

Get a list of all containers and their current location

Title	Get a list of all containers and their current location
Description and Use Case	Get a list of containers (their ID) and their current location including during transport. This will be used to track the location of all containers.
Transport Protocol Details	REST, endpoint: GET /containerLocation
Sent Body Example	-
Sent Body Description	-
Response Body Example	`[ { "container": "8a36d158-e545-404b-986f-1f752d173f08", "currentLocation": "c7c7482d-da86-44ec-ad35-713236251319", "distanceToGo": 2 } ]`

Response Body Description	• container: Id of requested containers • currentLocation: Id of the current container location • distanceToGo: distance to next location if traveling
Error Cases	• A hub-operator node was not found: 404 • Internal Server Error (should never happen only for debugging)

Get a list of all containers and their location history

Title	Get a list of all containers and their location history
Description and Use Case	Get a list of containers (their ID) and their location history. This will be used to track the containers and where they have been.
Transport Protocol Details	REST, endpoint: GET /locationHistory
Sent Body Example	-
Sent Body Description	-

Response Body Example	[
	{
	"container": "8a36d158-e545-404b-986f-1f752d173f08",
	"locationHistory": [
	"c7c7482d-da86-44ec-ad35-713236251319"
	]
	}
	]

Response Body Description	• container: Id of requested containers • locationHistory: Array of Ids of all location the container has been
Error Cases	• A hub-operator node was not found: 404 • Internal Server Error (should never happen only for debugging)

Messages & Communication:
(For further information see Documentation of the Messages & Communication at the end of the document.)

Publisher of:

- StartSignal: Sends this message if every MS has sent the message InstanceOnlineMessage to it. A requirement for this to work, is that the Hup Operator is online before the other MS.

Listener of:

- HubCapacityUpdate: reads out the capacity of the hubs to monitor it
- VehiclePositionUpdate: needs to listen this message to display the current positions of containers
- ContainerAtFinalDestination: uses this messages information to add to the containers history

- ContainerPositionUpdates: the HubOperator updates the container history according to information send
- InstanceOnlineMessage: needs to listen to this message to determine if every MS is online

With these messages all data can be collected to successfully provide the information needed for our WebUI and Database.

Status:
The basic Structure is implemented and the spring-boot web starter and the spring-boot gradle plugin were added as dependencies. The concrete implementation is still to implement. The database and the ORM mapper have to be created and instantiated.

Testing and presentation:

Testing strategy and techniques (Unit Tests):

Initially the individual components of the Message Framework will be tested using unit tests. The input/output message queues will be injected and will be filled with mocked messages to assess the components behaviour. For example, the MessageDelegator class will be injected one data structure for incoming messages, message callbacks and published messages respectively. It is therefore necessary that the MF classes all offer the possibility to inject the operating data structures and to not create them on their own. (dependency injection).
The Microservices might be a bit trickier in this regard, since they rely on asynchronous updates by the callbacks provided by the message framework. Individual functions of the callback execution of the MF can be mocked for testing purposes. Regarding the internal logic of the MS, e.g. routing, it is crucial that the single-responsibility principle is applied to all classes and methods.
We will use the JUnit testing framework because it widely used and will allow us to write tests and detect errors quickly and reliably. The only drawback of JUnit might be the lack of dependency tests.
For Mocking, the Mockito Framework will be used.

Testing strategy and techniques (Integration Tests):

With integration tests we face the problem of how to test an isolated Microservices. Since an isolated Microservice cannot properly execute its behaviour, we need to simulate the “exterior” of the tested Microservice. This shall be done for each Microservice.
Our integration test will have to run as a separate process and test the microservice only by exchanging messages through the Message Framework. The test actively must publish messages according to the use-case to be tested and assess the messages published by the microservice.
The Message Framework however, can not be tested this way. For that we need at least two test components/classes. One which sends/receives messages through the Message Framework and one which sends/receives them directly using sockets. This way errors/inconsistencies in the Messages Framework can be isolated more easily. The test-cases should have a pre-defined procedure so that the receiver can compare the behaviour of the message framework to the expected behaviour.
The testing of the two RESTful APIs will be done in an equal fashion.

Documentation of the Messages & Communication used by MS/MF

Title	Publish Connection Information of Hubs
Description and Use Case	At the beginning of the setup the Hubs need to generate the whole map to route containers. Therefore, Hubs publish the Hubs/Stations they are connected to and receive this information from the other hubs.
Transport Protocol Details	Provided by the MF, message type: HubConnectionInformation

Created and Sent Payload	{
	"hub": "123e4567-e89b-12d3-a456-556642440000",
	"connections": "Collection<NodeConnection>"
	}
	
	Node Connection is provided by MF {"hub": boolean, "distance": int, "UUID": UUID}

Created and Sent Payload Description	• hub: A unique id of the hub, length of UUID, type: UUID, mandatory • connections: Lists the different connections of hub, which specifies the distance and if the connection is to a hub or station as well as their UUID, unlimited length. Type: Collection, mandatory
Relevant Response	-
Error Cases	-

Title	New Container arrived at a station
Description and Use Case	If a Station (Source) gets a new container in, it has to inform the hubs about the new container so they can pick it up. This message is also processed internally by the MF to ensure that messages are bridged to the new microservice.
Transport Protocol Details	Provided by the MF, message type: NewContainerAtSource

Created and Sent Payload	{
	"containerInformation":
	{
	"id": "123e4567-e89b-12d3-a456-556642440000",
	"weight": "5"
	"destination": "123e4567-e89b-12d3-a456-556642440000"
	"source": "123e4567-e89b-12d3-a456-556642440000"
	"in_hub": "boolean"
	"currHub": "123e4567-e89b-12d3-a456-556642440000"
	}
	}

Created and Sent Payload Description	• ContainerInformation: contains information about the container (further explained in the following) • id: A unique id of the container, length of UUID, type: UUID, mandatory • Weight: Weight of the container, unlimited length, type: int, mandatory • Destination: Id of the destination (Station) where the container should be delivered to, length of UUID, type: UUID, mandatory

	• Source: Id of the source (Station) where the container was first injected, length of UUID, type: UUID, mandatory • In_hub: true if the container is currently in stored in a hub, false if it on the way and out for a delievery type: boolean, mandatory • CurrHub: id of the hub where the container is stored in. Boolean, optional
Relevant Response	-
Error Cases	• Destination id doesn't exist • Source id doesn't exist • Weight is too heavy and cannot be load on any vehicle

Title	Update of Hub Capacity and Vehicle Off Load Permission
Description and Use Case	Since the hubs have a capacity limit, vehicles may not be able to off load and have to wait. So, if a new Vehicle arrives, it waits till it gets the permission from the hub. The Hub Operator needs to monitor the Capacity of the hubs.
Transport Protocol Details	Provided by the MF, message type: HubCapacityUpdate
Created and Sent Payload	{ "capacity": "5123" "hub": "123e4567-e89b-12d3-a456-556642440000" "nextVehicleToLoad": "some UUID"
Created and Sent Payload Description	• Capacity: free capacity of the hub, unlimited length, type: int, mandatory • Hub: A unique id of the hub, length of UUID, type: UUID, mandatory • NextVehicleToLoad: A unique id of the vehicle, length of UUID, type: UUID, optional
Relevant Response	-
Error Cases	-

Title	Position Update of the Vehicle
Description and Use Case	The vehicle frequently sends out position updates, which the Hub Operator needs to monitor container locations and the container to update itself.
Transport Protocol Details	Provided by the MF, message type: VehiclePositionUpdate

Created and Sent Payload	{
	"inTravel": "true"
	"distanceToGo": "0.2"
	"vehicleId": "some UUID"
	"originHub": "some UUID"
	"targetHub": "some UUID"
	"containers": "a list of containers"
	
	}

Created and Sent Payload Description	• InTravel: is true if the vehicle is currently travelling to some destination, false if it has to wait at the target hub, type: boolean, mandatory • distanceToGo: percentage of the distance which the vehicle needs to arrive at targetHub, value between 0 and 1, type: double, mandatory • vehicleId: A unique id of the vehicle, length of UUID, type: UUID, mandatory • OriginHub: A unique id of the hub where the vehicle started its journey from, length of UUID, type: UUID, mandatory • TargetHub: A unique id of the hub which the vehicle travels to, length of UUID, type: UUID, mandatory • Containers: A list of container Information (for further information look at message "New Container arrived at a station"), length of UUID, type: List<ContainerInformation>, mandatory
Relevant Response	-
Error Cases	Vehicle id doesn't exist

Title	Order of a vehicle
Description and Use Case	If a hub sends out containers to another hub/end destination or picks up containers from a source. The Vehicle receives an order with the information what to do.
Transport Protocol Details	Provided by the MF, message type: VehicleOrder

Created and Sent Payload	{
	"targetHub": "123e4567-e89b-12d3-a456-556642440000"
	"containers": "List of container information"
	"orderType": "HubToHub"
	"distance": "int"
	}

Created and Sent Payload Description	• targetHub: A unique id of the hub from where the vehicle starts the journey, length of UUID, type: UUID, mandatory • containers: A list of container Information (for further information look at message "New Container arrived at a station"), length of UUID, type: List<ContainerInformation>, mandatory • orderType: is the type of the order the vehicle gets, type: EOrderType, mandatory • Distance: specifies the distance to the targetHub EOrderType is an Enum provided by MF and contains types: "StationPickUp", "HubToHub". "HubToStation"
Relevant Response	-
Error Cases	TargetHub id doesn't exist

Title	Container arrives at final destination

Description and Use Case	The vehicle sends out that the container was delivered to its final destination.
Transport Protocol Details	Provided by the MF, message type: ContainerAtFinalDestination

Created and Sent Payload	{
	"containerInformation": "contains some container Information"
	
	}

Created and Sent Payload Description	• ContainerInformation: (for further information look at message "New Container arrived at a station")
Relevant Response	-
Error Cases	-

Title	Vehicle arrives at a hub
Description and Use Case	If the vehicle arrives at the hub is has to inform the hub that is here.
Transport Protocol Details	Provided by the MF, message type: VehicleArrival

Created and Sent Payload	{
	"target": "123e4567-e89b-12d3-a456-556642440000"
	"origin": "123e4567-e89b-12d3-a456-556642440000"
	"containers": "List of some containers"
	"vehicleId": "some UUID"
	}

Created and Sent Payload Description	• Target: A unique id of the hub or station from where the vehicle has started its journey, length of UUID, type: UUID, mandatory • Origin: A unique id of the hub or station the vehicle has as its destination, length of UUID, type: UUID, mandatory • Containers: A list of container Information (for further information look at message "New Container arrived at a station"), length of UUID, type: List<ContainerInformation>, mandatory • VehicleId: A unique id of the vehicle, length of UUID, type: UUID, mandatory
Relevant Response	-
Error Cases	Target or origin hub id doesn't exist

Title	Container position update
Description and Use Case	Container information the container needs to update its location.

	This is also processed internally by the MF to ensure that Hubs and Vehicles always bridge to their current containers and them only.
Transport Protocol Details	Provided by the MF, message type: ContainerPositionUpdates

Created and Sent Payload	{
	"containerInformation": "contains some container Information"
	
	}

Created and Sent Payload Description	• ContainerInformation: (for further information look at message “New Container arrived at a station”)
Relevant Response	-
Error Cases	Container doesn’t exist

Title	Instance Online Message
Description and Use Case	Used initially by all Microservices to signal that they are online. The hub operator reads them and sends its own
Transport Protocol Details	Provided by the MF, message type: InstanceOnlineMessage

Created and Sent Payload	{
	"type": "type of MS"
	
	}

Created and Sent Payload Description	• type: The type of the MS which sends this message out, type: Enum of the MS type, mandatory
Relevant Response	-
Error Cases	

Title	Start Signal
Description and Use Case	Sent by the Hub Operator when all MS instances are online and signals them to start their operation.
Transport Protocol Details	Provided by the MF, message type: StartSignal
Created and Sent Payload	None
Created and Sent Payload Description	-
Relevant Response	-
Error Cases	

Title	Container Handover
Description and Use Case	Sent whenever a Container is handed over (by the Vehicle/Hub/Station handing it over) and used only be the Message Framework to configure the new connected MS for bridging.
Transport Protocol Details	Provided by the MF, message type: ContainerHandover

Created and Sent Payload	{
	"source": "source UUID"
	"target": "target UUID"
	}

Created and Sent Payload Description	• source: UUID of the source, type: UUID, mandatory • target: UUID of the target, type: UUID, mandatory • containerInformation: includes Information about the Container, mandatory
Relevant Response	-
Error Cases	