openTCS
User's Guide

The openTCS developers

Version 5.14.0

Table of Contents

Chapter 1. Introduction

1.1. Purpose of the software

openTCS (short for *open Transportation Control System*) is a free control system software for coordinating fleets of automated guided vehicles (AGVs) and mobile robots, e.g. in production plants. It should generally be possible to control any automatic vehicle with communication capabilities with it, but AGVs are the main target.

openTCS controls vehicles independent of their specific characteristics like navigation principle/track guidance system or load handling device. It can manage vehicles of different types (and performing different tasks) at the same time.

openTCS itself is not a complete product you use out-of-the-box to control AGVs with. Primarily, it is a framework/an implementation of the basic data structures and algorithms (routing of vehicles, dispatching orders to them, managing the fleet's traffic) needed for running an AGV system with more than one vehicle. It tries to be as generic as possible to allow interoperation with vehicles of practically any vendor.

As a consequence, it is usually necessary to at least create and plug in a vehicle driver (called *communication adapter* in openTCS-speak) that translates between the abstract interface of the openTCS kernel and the communication protocol your vehicle understands. (Such vehicle drivers are similar to device drivers in operating systems, in a way.) Depending on your needs, it might also be necessary to adapt algorithms or add project-specific strategies.

1.2. System requirements

openTCS does not come with any specific hardware requirements. CPU power and RAM capacity highly depend on the use case, e.g. the size and complexity of the driving course and the number of vehicles managed. Some kind of networking hardware — in most cases simply a standard Ethernet controller — is required for communicating with the vehicles (and possibly other systems, like a warehouse management system).

To run openTCS, a Java Runtime Environment (JRE) version 13 is required. (The directory `bin` of the installed JRE, for example `C:\Program Files\AdoptOpenJDK\jdk-13.0.2.8-hotspot\bin`, should be included in the enviroment variable PATH to be able to use the included start scripts.)

> Due to a limitation in a software library used by openTCS (namely: Docking Frames), some JREs are currently not compatible with openTCS. This is true e.g. for the JRE provided by Oracle. The recommended JRE to use is the one provided by the Adoptium project.

1.3. Licensing

openTCS is being maintained by the openTCS team at the Fraunhofer Institute for Material Flow and Logistics.

The openTCS source code is licensed under the terms of the MIT License. Please note that openTCS is distributed without any warranty - without even the implied warranty of merchantability or fitness for a particular purpose. Please refer to the license (`LICENSE.txt`) for details.

1.4. Further documentation

If you intend to extend and customize openTCS, please also see the Developer's Guide and the JavaDoc documentation that is part of the openTCS distribution.

1.5. Support

Please note that, while Fraunhofer IML is happy to be able to release openTCS to the public as free software and would like to see it used and improved continuously, the development team cannot provide unlimited free support for it.

If you have technical/support questions, please post them on the project's discussion forum, where the community and the developers involved will respond as time permits. You can find the discussion forum at https://github.com/openTCS/opentcs/discussions. Please remember to include enough data in your problem report to help the developers help you, e.g.:

- The applications' log files, contained in the subdirectory `log/` of the kernel, kernel control center, model editor and operations desk application
- The plant model you are working with, contained in the subdirectory `data/` of the kernel and/or model editor application

Chapter 2. System overview

2.1. System components and structure

openTCS consists of the following components running as separate processes and working together in a client-server architecture:

- Kernel (server process), running vehicle-independent strategies and drivers for controlled vehicles

- Clients

  ◦ Model editor for modelling the plant model

  ◦ Operations desk for visualizing the plant model during plant operation

  ◦ Kernel control center for controlling and monitoring the kernel, e.g. providing a detailed view of vehicles/their associated drivers

  ◦ Arbitrary clients for comunicating with other systems, e.g. for process control or warehouse management

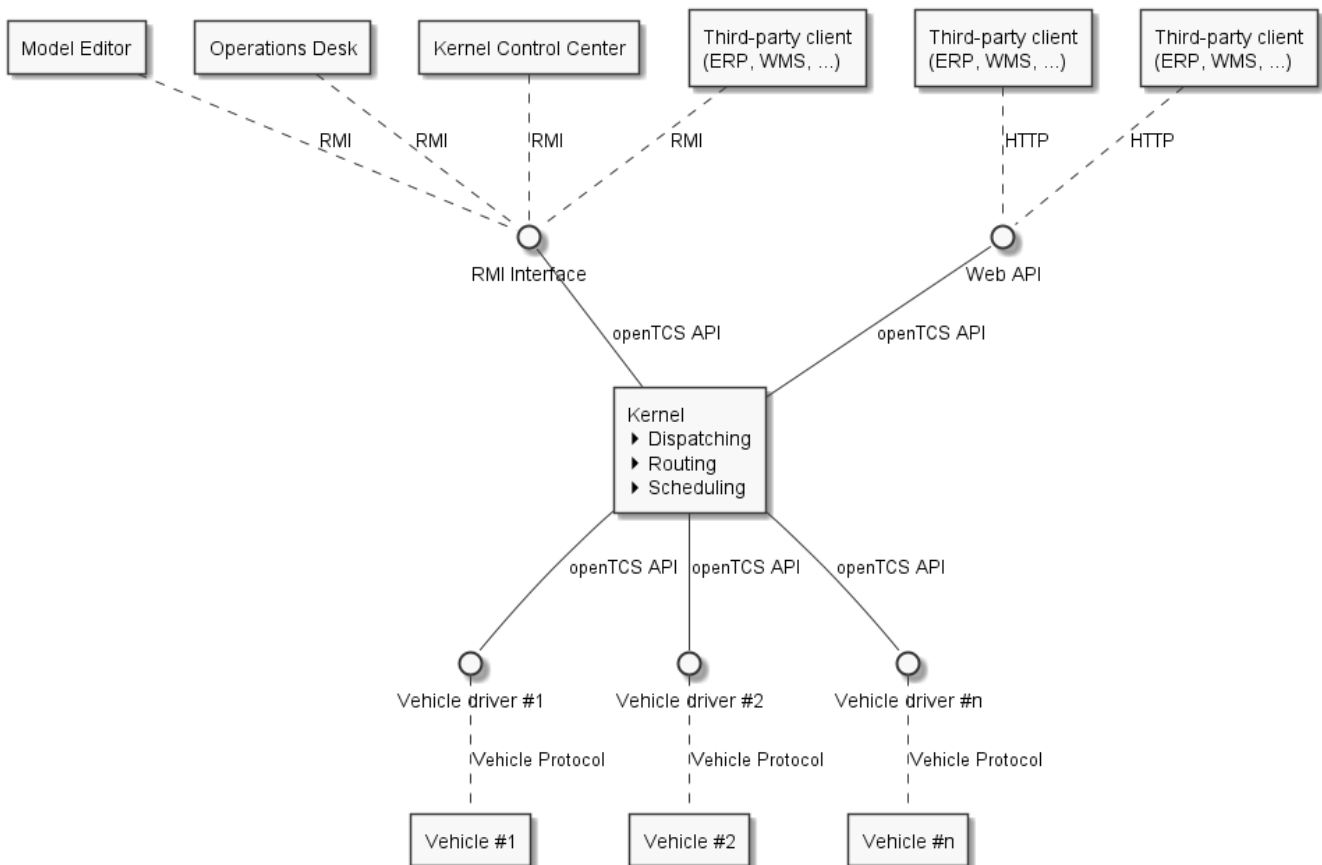*Figure 1. openTCS system overview*

The purpose of the openTCS kernel is to provide an abstract driving model of a transportation system/plant, to manage transport orders and to compute routes for the vehicles. Clients can communicate with this server process to, for instance, modify the plant model, to visualize the driving course and the processing of transport orders and to create new transport orders.

Three major strategy modules within the kernel implement processing of transport orders:

- A dispatcher that decides which transport order should be processed by which vehicle. Additionally, it needs to decide what vehicles should do in certain situation, e.g. when there aren't any transport orders or when a vehicle is running low on energy.
- A router which finds optimal routes for vehicles to reach their destinations.
- A scheduler that manages resource allocations for traffic management, i.e. to avoid vehicles crashing into each other.

The openTCS distribution comes with default implementations for each of these strategies. These implementations can be easily replaced by a developer to adapt to environment-specific requirements.

The driver framework that is part of the openTCS kernel manages communication channels and associates vehicle drivers with vehicles. A vehicle driver is an adapter between kernel and vehicle and translates each vehicle-specific communication protocol to the kernel's internal communication schemes and vice versa. Furthermore, a driver may offer low-level functionality to the user via the kernel control center client, e.g. manually sending telegrams to the associated vehicle. By using suitable vehicle drivers, vehicles of different types can be managed simultaneously by a single openTCS instance.

The model editor client that is part of the openTCS distribution allows editing of plant models, which can be loaded into the kernel. This includes, for instance, the definition of load-change stations, driving tracks and vehicles.

The operations desk client that is part of the openTCS distribution is used to display the transportation system's general state and any active transport processes, and to create new transport orders interactively.

The kernel control center client that is part of the openTCS distribution allows controlling and monitoring the kernel. Part of that is assigning vehicle drivers to vehicles and controlling them by enabling the communication and monitoring them by displaying vehicle state information, for instance.

Other clients, e.g. to control higher-level plant processes, can be implemented and attached. For Java clients, the openTCS kernel provides an interface based on Java RMI (Remote Method Invocation). Additionally, openTCS provides a web API for creating and withdrawing transport orders and retrieving transport order status updates.

2.2. Plant model elements

In openTCS, a plant model consists of a set of the following elements. The attributes of these elements that are relevant for the plant model, e.g. the coordinates of a point or the length of a path, can be edited using the model editor client.

2.2.1. Point

Points are logical mappings of discrete vehicle positions in the driving course. In plant operation

mode, vehicles are ordered (and thus move) from one point to another in the model. A point carries the following attributes:

- A *type*, which is one of these three:

  - *Halt position*: Indicates a position at which a vehicle may halt temporarily while processing an order, e.g. for executing an operation. The vehicle is expected to report in when it arrives at such a position. It may not remain here for longer than necessary, though. Halt position is the default type for points when modelling with the model editor client.

  - *Reporting position* (deprecated, scheduled for removal in openTCS 6.0): Indicates a position at which a vehicle is expected to report in *only*. Vehicles will not be ordered to a reporting position, and halting or even parking at such a position is not allowed. Therefore a route that only consists of reporting points will be unroutable because the vehicle is not able to halt at any position.

  - *Park position*: Indicates a position at which a vehicle may halt for longer periods of time when it is not processing orders. The vehicle is also expected to report in when it arrives at such a position.

- A *position*, i.e. the point's coordinates in the plant's coordinate system.

- A *vehicle orientation angle*, which expresses the vehicle's assumed/expected orientation while it occupies the point.

> In openTCS, an angle of 0 degrees is at the 3 o'clock position, and a positive value indicates a counter-clockwise rotation.

2.2.1.1. Layout coordinates vs model coordinates

A point has two sets of coordinates: layout coordinates and model coordinates. The layout coordinates are merely intended for the graphical presentation in the model editor and operations desk clients, while the model coordinates are data that a vehicle driver could potentially use or send to the vehicle it communicates with (e.g. if the vehicle needs the exact coordinates of a destination point for navigation). Both coordinate sets are not tied to each other per se, i.e. they may differ. This is to allow coordinates that the system works with internally to be different from the presentation; for example, you may want to provide a distorted view on the driving course simply because some paths in your plant are very long and you mainly want to view all points/locations closely together. Dragging points and therefore changing their position in the graphical presentation only affects the corresponding layout coordinates.

To synchronize the layout coordinates with the model coordinates or the other way around you have two options:

- Select [**Actions › Copy model values to layout**] or [**Actions › Copy layout values to model**] to synchronize them globally.

- Select a single layout element, right click it and select [**Context menu › Copy model values to layout**] or [**Context menu › Copy layout values to model**] to synchronize them only for the selected element.

2.2.2. Path

Paths are connections between points that are navigable for vehicles. A path's main attributes, next to its source and destination point, are:

- Its *length*, which may be a relevant information for a vehicle in plant operation mode. Depending on the router configuration, it may also be used for computing routing costs/finding an optimal route to a destination point.

- A *maximum velocity* and *maximum reverse velocity*, which may be a relevant information for a vehicle in plant operation mode. Depending on the router configuration, it may also be used for computing routing costs/finding an optimal route to a destination point.

- A *locked* flag, which, when set, tells the router that the path may not be used when computing routes for vehicles.

- A sequence of *peripheral operations* describing operations that are to be perfomed by peripheral devices (in their given order) when a vehicle traverses the path.

2.2.2.1. Peripheral operation

A peripheral operation's attributes are:

- A reference to the *location* representing the peripheral device by which the operation is to be performed — see Location.

- The actual *operation* to be performed by the peripheral device.

- An *execution trigger* defining the moment at which the operation is to be performed. The supported values are:

  - `BEFORE_MOVEMENT`: The execution of the operation should be triggered *before* a vehicle traverses the path.

  - `AFTER_MOVEMENT`: The execution of the operation should be triggered *after* a vehicle has traversed the path.

- A *completion required* flag, which, when set, requires the operation to be completed to allow a vehicle to continue driving. This flag works in combination with the execution trigger. With the `BEFORE_MOVEMENT` execution trigger and the completion required flag set to `true`, a vehicle has to wait at the path's source point until the operation is completed. With the `AFTER_MOVEMENT` execution trigger and the completion required flag set to `true`, a vehicle has to wait at the path's destination point until the operation is completed.

2.2.3. Location

Locations are markers for points at which vehicles may execute special operations (load or unload cargo, charge their battery etc.). A location's attributes are:

- Its *type*, basically defining which operations are allowed at the location — see Location type.

- A set of *links* to points that the location can be reached from. To be of any use for vehicles in the plant model, a location needs to be linked to at least one point.

- A *locked* flag, which, when set, tells the dispatcher that transport orders requiring an operation

at the location may not be assigned to vehicles.

Additionally, locations can map peripheral devices for the purpose of communicating with them and allowing vehicles to interact with them (e.g. opening/closing fire doors along paths). See Adding and configuring peripheral devices for details on how to add and configure peripheral devices.

2.2.4. Location type

Location types are abstract elements that group locations. A location type has only two relevant attributes:

- A set of *allowed/supported vehicle operations*, defining which operations a vehicle may execute at locations of this type.

- A set of *allowed/supported peripheral operations*, defining which operations peripheral devices mapped to locations of this type may execute.

2.2.5. Vehicle

Vehicles map physical vehicles for the purpose of communicating with them and visualizing their positions and other characteristics. A vehicle provides the following attributes:

- A *critical energy level*, which is the threshold below which the vehicle's energy level is considered critical. This value may be used at plant operation time to decide when it is crucial to recharge a vehicle's energy storage.

- A *good energy level*, which is the threshold above which the vehicle's energy level is considered good. This value may be used at plant operation time to decide when it is unnecessary to recharge a vehicle's energy storage.

- A *fully recharged energy level*, which is the threshold above which the vehicle is considered being fully recharged. This value may be used at plant operation time to decide when a vehicle should stop charging.

- A *sufficiently recharged energy level*, which is the threshold above which the vehicle is considered sufficiently recharged. This value may be used at plant operation time to decide when a vehicle may stop charging.

- A *maximum velocity* and *maximum reverse velocity*. Depending on the router configuration, it may be used for computing routing costs/finding an optimal route to a destination point.

- An *integration level*, indicating how far the vehicle is currently allowed to be integrated into the system. A vehicle's integration level can only be adjusted with the operations desk client, not with the model editor client. A vehicle can be

  - ...*ignored*: The vehicle and its reported position will be ignored completely, thus the vehicle will not be displayed in the operatiosn desk. The vehicle is not available for transport orders.

  - ...*noticed*: The vehicle will be displayed at its reported position in the operations desk, but no resources will be allocated in the system for that position. The vehicle is not available for transport orders.

  - ...*respected*: The resources for the vehicle's reported position will be allocated. The vehicle is

not available for transport orders.

- ○ *...utilized*: The vehicle is available for transport orders and will be utilized by the openTCS.

- A *paused* flag, indicating whether the vehicle is currently paused or not. A vehicle that is paused is supposed not to move/operate. In case it is currently moving when its paused flag is set, it is expected to stop as soon as possible. Some vehicle types may not support stopping before reaching their movement commands' destination. In such cases, openTCS will still ensure no further movement commands are sent to vehicles as long as they are paused.

- A set of *allowed transport order types*, which are strings used for filtering transport orders (by their type) that are allowed to be assigned to the vehicle. Also see Transport order.

- A *route color*, which is the color used for visualizing the route the vehicle is taking to its destination.

2.2.6. Block

Blocks (or block areas) are areas for which special traffic rules may apply. They can be useful to prevent deadlock situations, e.g. at path intersections or dead ends. A block has two relevant attributes:

- A set of *members*, i.e. resources (points, paths and/or locations) that the block is composed of.

- A *type*, which determines the rules for entering a block:

  - ○ *Single vehicle only*: The resources aggregated in this block can only be used by a single vehicle at the same time. This is the default type for blocks when modelling with the model editor client.

  - ○ *Same direction only*: The resources aggregated in this block can be used by multiple vehicles at the same time, but only if they traverse the block in the same direction.

    > The direction in which a vehicle traverses a block is determined using the first allocation request containing resources that are part of the block — see Default scheduler. For the requested resources (usually a point and a path) the path is checked for a property with the key `tcs:blockEntryDirection`. The property's value may be an arbitrary character string (including the empty string). If there is no such property the path's name is being used as the direction.

2.2.7. Layer

Layers are abstract elements that group points, paths, locations and links. They can be useful for modelling complex plants and dividing plant sections into logical groups (e.g. floors in a multi-floor plant). A layer has the following properties:

- An *active* flag, which indicates whether a layer is currently set as the active (drawing) layer. There can only be one active layer at a time. This property is shown only in the model editor client.

- A *visible* flag, which indicates whether a layer is shown or hidden. When a layer is hidden, the model elements it contains are not displayed.

- A descriptive *name*.

- A *group,* that the layer is assigned to — see Layer group. A layer can only be assigned to one layer group at a time.

- A *group visible* flag, which indicates whether the layer group the layer is assigned to is shown or hidden — see Layer group.

In addition to the properties listed above, layers also have an ordinal number (which is not displayed) that defines the order of the layers in relation to each other. The order of the layers is represented by the order of the entries in the "Layers" table in the Model Editor and the Operations Desk clients. The topmost entry corresponds to the topmost layer (which is displayed above all other layers) and the bottommost entry corresponds to the bottommost layer (which is displayed below all other layers).

2.2.8. Layer group

Layer groups are abstract elements that group layers. A layer group has the following properties:

- A descriptive *name.*

- A *visible* flag, which indicates whether the layer group is shown or hidden. When a layer group is hidden, the model elements contained in all layers assigned to it are not displayed. The visibility state of a layer group doesn't affect the visibility state of the layers assigned to it.

2.3. Plant operation elements

Transport orders and order sequences are elements that are available only at plant operation time. Their attributes are primarily set when the respective elements are created.

2.3.1. Transport order

A transport order is a parameterized sequence of movements and operations to be processed by a vehicle. When creating a transport order, the following attributes can be set:

- A sequence of *destinations* that the processing vehicle must process (in their given order). Each destination consists of a location that the vehicle must travel to and an operation that it must perform there.

- An optional *deadline,* indicating when the transport order is supposed to have been processed.

- An optional *type,* which is a string used for filtering vehicles that may be assigned to the transport order. A vehicle may only be assigned to a transport order if the order's type is in the vehicle's set of allowed order types. (Examples for potentially useful types are `"Transport"` and `"Maintenance"`.)

- An optional *intended vehicle,* telling the dispatcher to assign the transport order to the specified vehicle instead of selecting one automatically.

- An optional set of *dependencies,* i.e. references to other transport orders that need to be processed before the transport order. Dependencies are transitive, meaning that if order A depends on order B and order B depends on order C, C must be processed first, then B, then A. As a result, dependencies are a means to impose an order on sets of transport orders. (They do not, however, implicitly require all the transport orders to be processed by the same vehicle.

This can optionally be achieved by also setting the *intended vehicle* attribute of the transport orders.) The following image shows an example of dependencies between multiple transport orders:
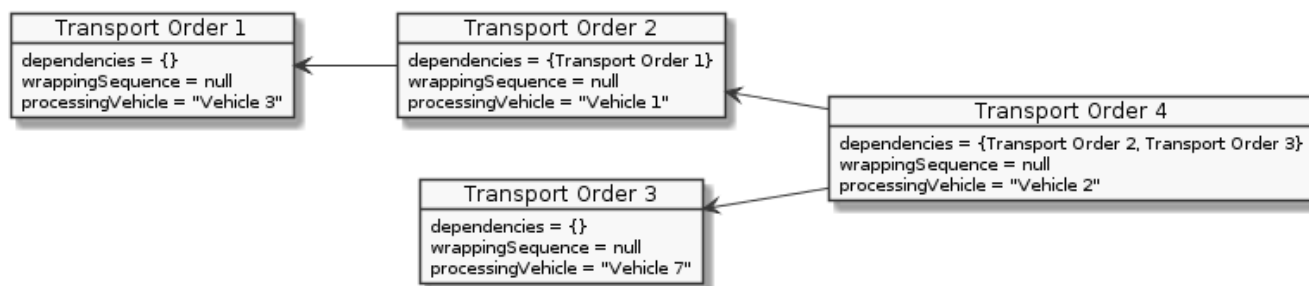
Figure 2. Transport order dependencies

2.3.2. Order sequence

> **ℹ** The operations desk application currently does not provide a way to create order sequences. They can only be created programmatically, using dedicated clients that are not part of the openTCS distribution.

An order sequence describes a process spanning multiple transport orders which are to be executed subsequently — in the exact order defined by the sequence — by a single vehicle. Once a vehicle is assigned to an order sequence, it may not process transport orders not belonging to the sequence, until the latter is finished.

Order sequences are useful when a complex process to be executed by one and the same vehicle cannot be mapped to a single transport order. This can be the case, for instance, when the details of some steps in the process become known only after processing previous steps.

An order sequence carries the following attributes:

- A sequence of *transport orders*, which may be extended as long the complete flag (see below) is not set, yet.

- A *complete* flag, indicating that no further transport orders will be added to the sequence. This cannot be reset.

- A *failure fatal* flag, indicating that, if one transport order in the sequence fails, all orders following it should immediately be considered as failed, too.

- A *finished* flag, indicating that the order sequence has been processed (and the vehicle is not bound to it, anymore). An order sequence can only be marked as finished if it has been marked as complete before.

- An optional *type* — see Transport order. An order sequence and all transport orders it contains (must) share the same type.

- An optional *intended vehicle*, telling the dispatcher to assign the order sequence to the specified vehicle instead of selecting one automatically. If set, all transport orders added to the order sequence must carry the same intended vehicle value.
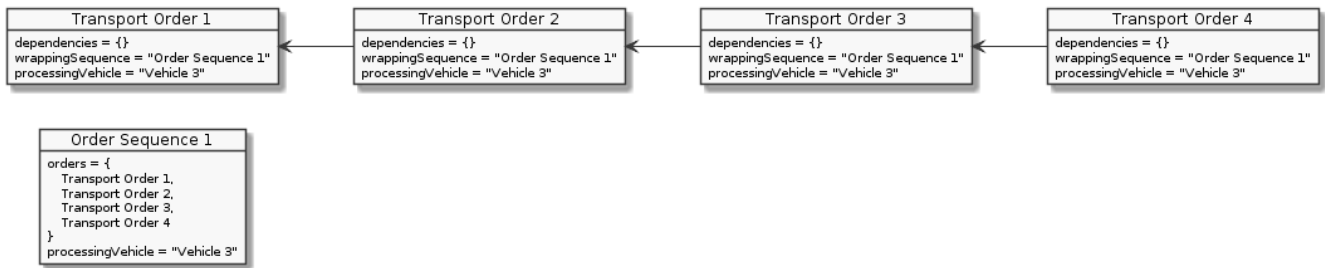
Figure 3. An order sequence

2.3.3. Peripheral job

A peripheral job describes an operation to be performed by a peripheral device. A peripheral job carries the following attributes:

- An *operation* to be performed by a peripheral device — see Peripheral operation.

- A *reservation token* that may be used to reserve a peripheral device. A peripheral device that is reserved for a specific token can only process jobs which match that reservation token — see Reservation token.

- An optional *related vehicle* referencing the vehicle by which the peripheral job was created.

- An optional *related transport order* referencing the transport order in which context the peripheral job was created.

2.4. Common element attributes

2.4.1. Unique name

Every plant model and plant operation element has a unique name identifying it in the system, regardless of what type of element it is. Two elements may not be given the same name, even if e.g. one is a point and the other one is a transport order.

2.4.2. Generic properties

In addition to the listed attributes, it is possible to define arbitrary properties as key-value pairs for all driving course elements, which for example can be read and evaluated by vehicle drivers or client software. Both the key and the value can be arbitrary character strings. For example, a key-value pair `"IP address":"192.168.23.42"` could be defined for a vehicle in the model, stating which IP address is to be used to communicate with the vehicle; a vehicle driver could now check during runtime whether a value for the key `"IP address"` was defined, and if yes, use it to automatically configure the communication channel to the vehicle. Another use for these generic attributes can be vehicle-specific actions to be executed on certain paths in the model. If a vehicle should, for instance, issue an acoustic warning and/or turn on the right-hand direction indicator when currently on a certain path, attributes with the keys `"acoustic warning"` and/or `"right-hand direction indicator"` could be defined for this path and evaluated by the respective vehicle driver.

Chapter 3. Operating the system

To create or to edit the plant model of a transport system, use the Model Editor application.

As a graphical frontend for a transportation control system based on an existing plant model, use the Operations Desk application. Note that the Operations Desk application always requires a running openTCS kernel that it can connect to.

Starting an application is done by executing the respective Unix shell script (*.sh) or Windows batch file (*.bat).

3.1. Constructing a new plant model

These instructions demonstrate how a new plant model is created and filled with driving course elements so that it can eventually be used for plant operation.

3.1.1. Starting the Model Editor

1. Launch the Model Editor (startModelEditor.bat/.sh).

2. The Model Editor will start with a new, empty model, but you can also load a model from a file (**[File › Load Model]**) or the current kernel model (**[File › Load current kernel model]**). The latter option requires a running kernel that the Model Editor client can connect to.

3. Use the graphical user interface of the Model Editor client to create an arbitrary driving course for your respective application/project. How you can add elements like points, paths and vehicles to your driving course is explained in detail in the following section. Whenever you want to start over, select **[File › New Model]** from the main menu.

3.1.2. Adding elements to the plant model
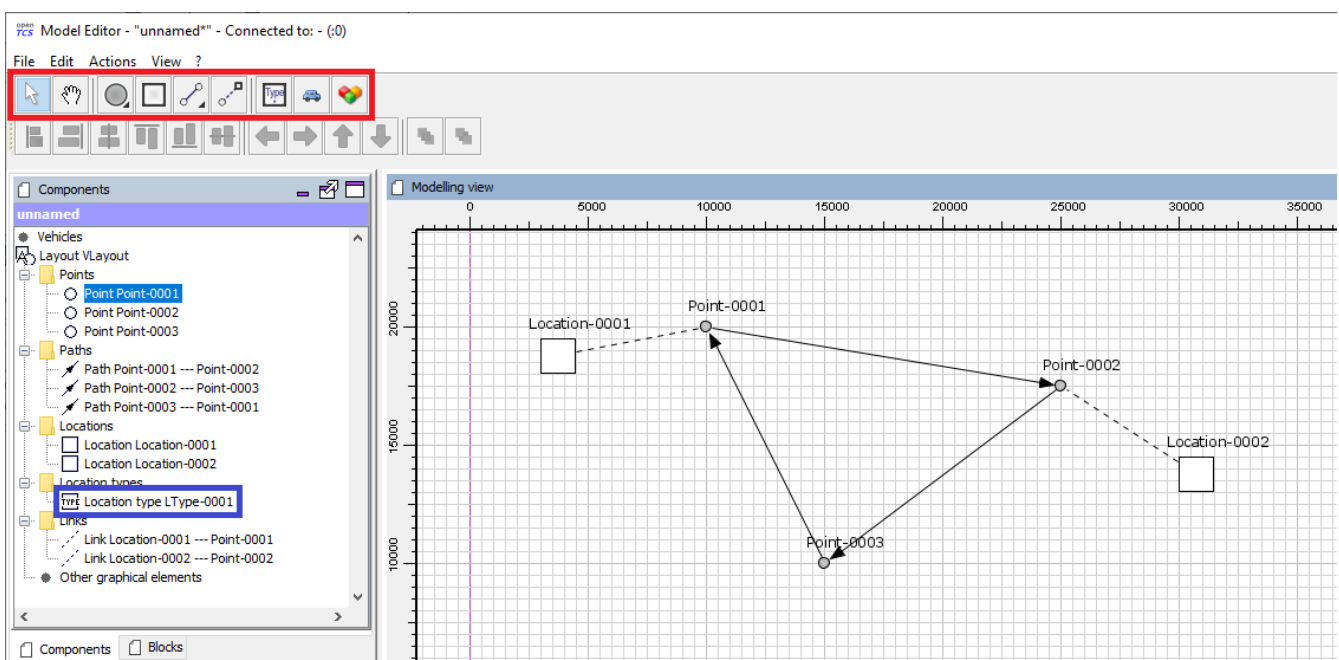
Figure 4. Control elements in the Model Editor client

1. Create three points by selecting the point tool from the driving course elements toolbar (see red frame in the screenshot above) and click on three positions on the drawing area.

2. Link the three points with paths to a closed loop by

   a. selecting the path tool from the driving course elements toolbar.

   b. clicking on a point, dragging the path to the next point and releasing the mouse button there.

3. Create two locations by selecting the location tool from the driving course elements toolbar and clicking on any two free positions on the drawing area. As a location type does not yet exist in the plant model, a new one is created implicitly when creating the first location, which can be seen in the tree view to the left of the drawing area.

4. Link the two locations with (different) points by

   a. selecting the link tool from the driving course elements toolbar.

   b. clicking on a location, dragging the link to a point and releasing the mouse button.

5. Create a new vehicle by clicking on the vehicle button in the driving course elements toolbar.

6. Define the allowed operations for vehicles at the newly created locations by

   a. selecting the locations' type in the tree view to the left of the drawing area (see blue frame in the screenshot above).

   b. clicking the value cell labelled **[Supported vehicle operations]** in the property window below the tree view.

   c. entering the allowed operations as arbitrary text in the dialog shown, for instance `"Load cargo"` and `"Unload cargo"`.

   d. Optionally, you can choose a symbol for locations of the selected type by editing the property `"Symbol"`.

   > ⊗ You will not be able to create any transport orders and assign them to vehicles when operating the system unless you create locations in your plant model, link these locations to points in the driving course and define the operations that vehicles may execute with the respective location types.

3.1.2.1. Adding and configuring peripheral devices

As described in Plant model elements, locations can be used to map peripheral devices. This means that, after executing the steps described above, there are now two locations available in the plant model that can potentially be used to integrate two peripheral devices. To integrate an exemplary peripheral device, the following additional steps are required:

1. Associate a location with a peripheral device by

   a. selecting the location in the tree view to the left of the drawing area (e.g. "Location-0001" in the screenshot above).

   b. clicking the value cell labelled **[Miscellaneous]** in the property window below the tree view.

   c. adding a key-value pair with the key `tcs:loopbackPeripheral` and an empty value.

2. Define the allowed operations for the peripheral device associated with this location by

   a. selecting the locations' type in the tree view to the left of the drawing area (e.g. "LType-0001" in the screenshot above).

   b. clicking the value cell labelled **[Supported peripheral operations]** in the property window below the tree view.

   c. entering the allowed operations as arbitrary text in the dialog shown, for instance `"Open door"` and `"Close door"`.

3. Optionally, define peripheral operations on paths by

   a. selecting a path in the tree view to the left of the drawing area (e.g. "Point-0001 --- Point-0002" in the screenshot above).

   b. clicking the value cell labelled **[Peripheral operations]** in the property window below the tree view.

   c. configuring and adding peripheral operations via the dialog shown.

> **ℹ** The steps above describe the process of associating a location with a virtual peripheral device that is controlled by the loopback peripheral driver. Unlike vehicles, which don't require any additional configuration for the loopback vehicle driver to be assigned to them, locations specifically require the aforementioned property for the peripheral loopback driver to be assigned to them.

> **!** You will not be able to create any peripheral jobs and assign them to peripheral devices when operating the system unless you create locations in your plant model, associate these locations with peripheral devices and define the operations that peripheral devices may execute with the respective location types.

3.1.3. Working with layers and layer groups

In addition to adding elements to the plant model, the Model Editor allows plant models to be created with different layers and layer groups. For more details on the properties of layers and layer groups see Layer and Layer group.

> **ℹ** With the Operations Desk application it's only possible to show/hide layers and layer groups, but not to manipulate them.

3.1.3.1. Layers

A layer groups points, paths, locations and links and allows the driving course elements it contains to be shown or hidden on demand. Layers can be created, removed and edited using the panel shown in the screenshot below (see red frame). There are a few things to keep in mind when working with layers:

- There always has to be at least one layer.

- When adding a new layer, it always becomes the active layer.

- Removing a layer results in the driving course elements it contains to be removed as well.

- When adding model elements (i.e. points, paths, etc.) they are always placed on the active layer.

- Links (between locations and points) are always placed on the same layer the respective location is placed on, regardless of the active layer.

- When performing "copy & paste", "cut & paste" or "duplicate" operations on driving course elements, the copies are always placed on the active layer.

> In the Operations Desk application the visibility of layers (and layer groups) also affects whether vehicle elements are displayed in the plant model or not. Vehicle elements inherit the visibility of the point at which they are reported. If a vehicle is reported at a point that is part of a hidden layer (or layer group), the vehicle element is not displayed in the plant model.
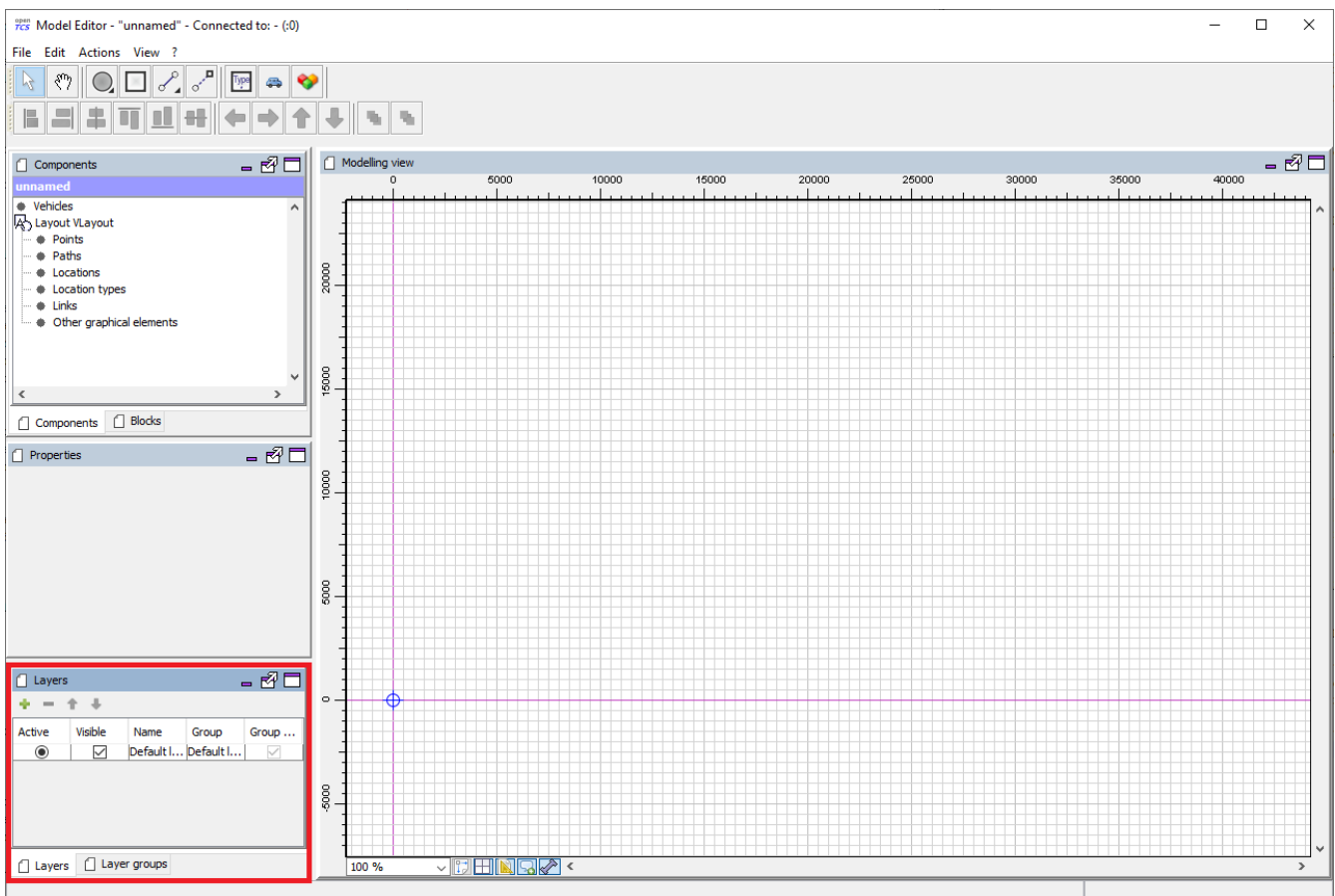
Figure 5. Layers panel (Toolbar buttons: Add layer, Remove (selected) layer, Move (selected) layer up, Move (selected) layer down)

3.1.3.2. Layer groups

A layer group groups, as the name implies, one or more layers and allows multiple layers to be shown or hidden at once. Layer groups can be created, removed and edited using the panel shown in the screenshot below (see red frame). There are a few things to keep in mind when working with layer groups:

- There always has to be at least one layer group.

- Removing a layer group results in all layers assigned to it to be removed as well.
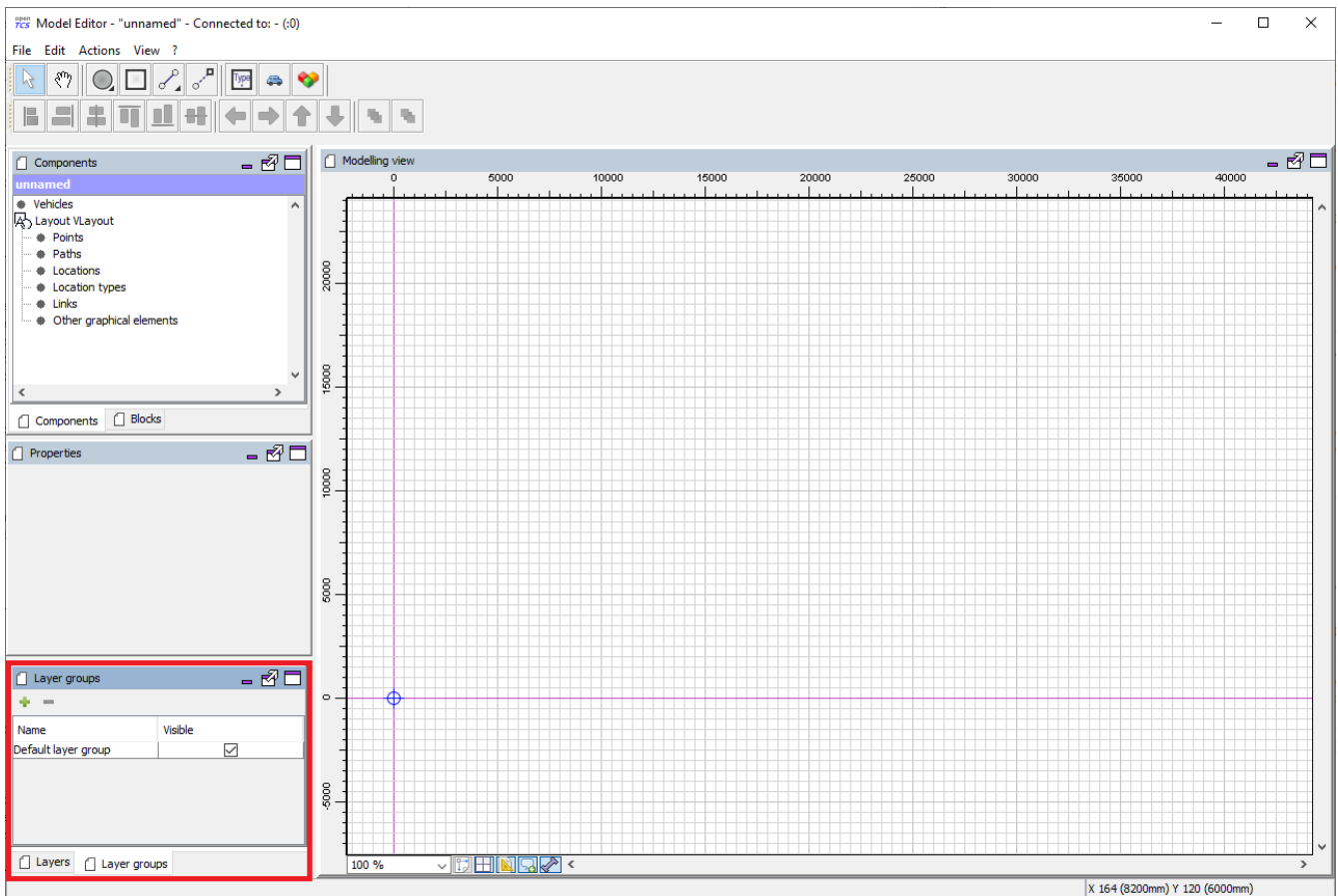
Figure 6. Layer groups panel (Toolbar buttons: Add layer group, Remove (selected) layer group)

3.1.4. Saving the plant model

You have two options to save the model: on your local hard drive or in a running kernel instance the Model Editor can connect to.

3.1.4.1. Saving the model locally

Select **[File › Save Model]** or **[File › Save Model As...]** and enter a file name for the model.

3.1.4.2. Loading the model into a running kernel

Select **[File › Upload model to kernel]** and your model will be loaded into the kernel, letting you use it for operating a plant. This, though, requires you to save it locally first. Note that any model that was previously loaded in the kernel will be replaced, as the kernel can only store a single model at a time.

3.2. Operating the plant

These instructions explain how the newly created model that was loaded into the kernel can be used for plant operation, how vehicle drivers are used and how transport orders can be created and processed by a vehicle.

3.2.1. Starting components for system operation
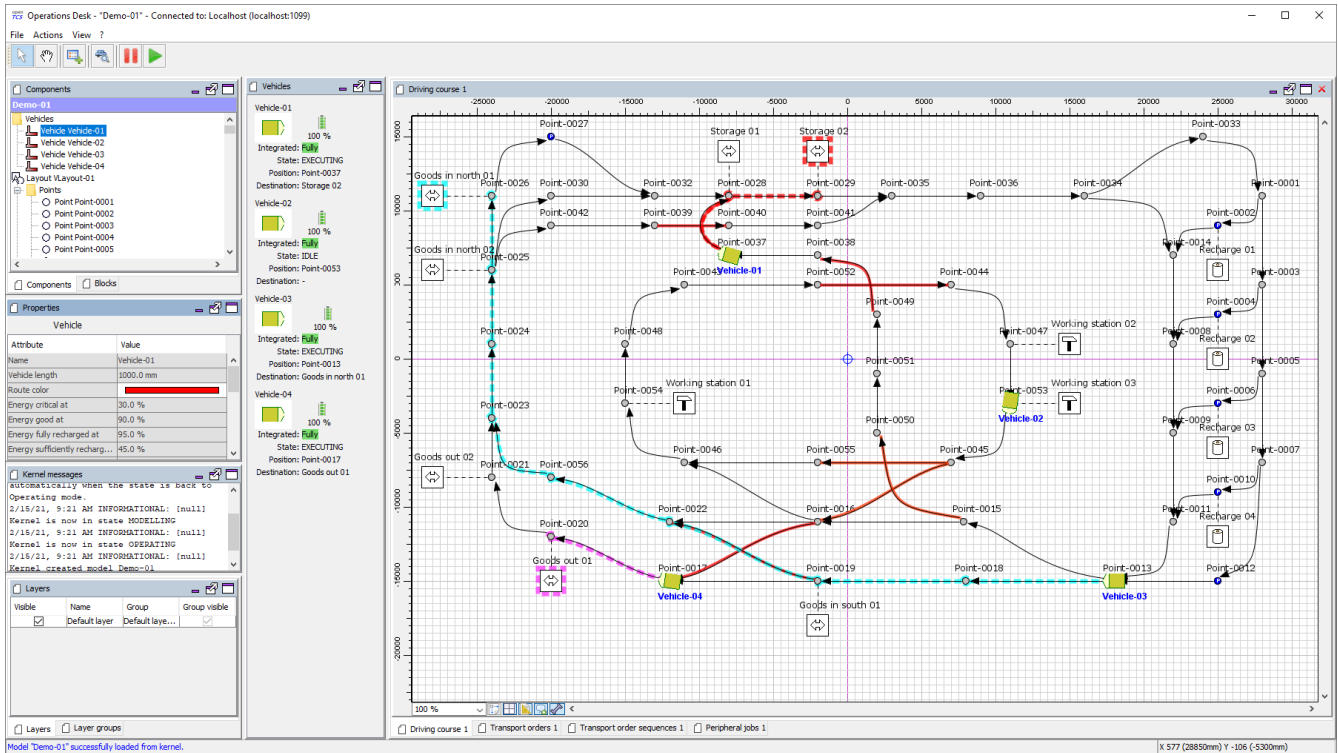
Figure 7. Operations Desk application displaying plant model

1. Start the kernel (`startKernel.bat/.sh`).

   a. If this is your first time running the kernel, you need to load an existing plant model from the Model Editor into the kernel first. (See Loading the model into a running kernel).

2. Start the Kernel Control Center application (`startKernelControlCenter.bat/.sh`)

3. Start the Operations Desk application (`startOperationsDesk.bat/.sh`)

4. In the Kernel Control Center, select the **[Vehicle driver]** tab. Then select, configure and start driver for each vehicle in the model.

   a. The list on the left-hand side of the window shows all vehicles in the chosen model.

   b. A detailed view for a vehicle can be seen on the right-hand side of the driver panel after double-clicking on the vehicle name in the list. The specific design of this detailed view depends on the driver associated with the vehicle. Usually, status information sent by the vehicle (e.g. current position and mode of operation) is displayed and low-level settings (e.g. for the vehicle's IP address) are provided here.

   c. Right-clicking on the list of vehicles shows a popup menu that allows to attach drivers to selected vehicles.

   d. For a vehicle to be controlled by the system, a driver needs to be attached to the vehicle and enabled. (For testing purposes without real vehicles that could communicate with the system, the so-called loopback driver can be used, which provides a virtual vehicle that roughly simulates a real one.) How you attach and enable a vehicle driver is explained in detail in Configuring vehicle drivers.
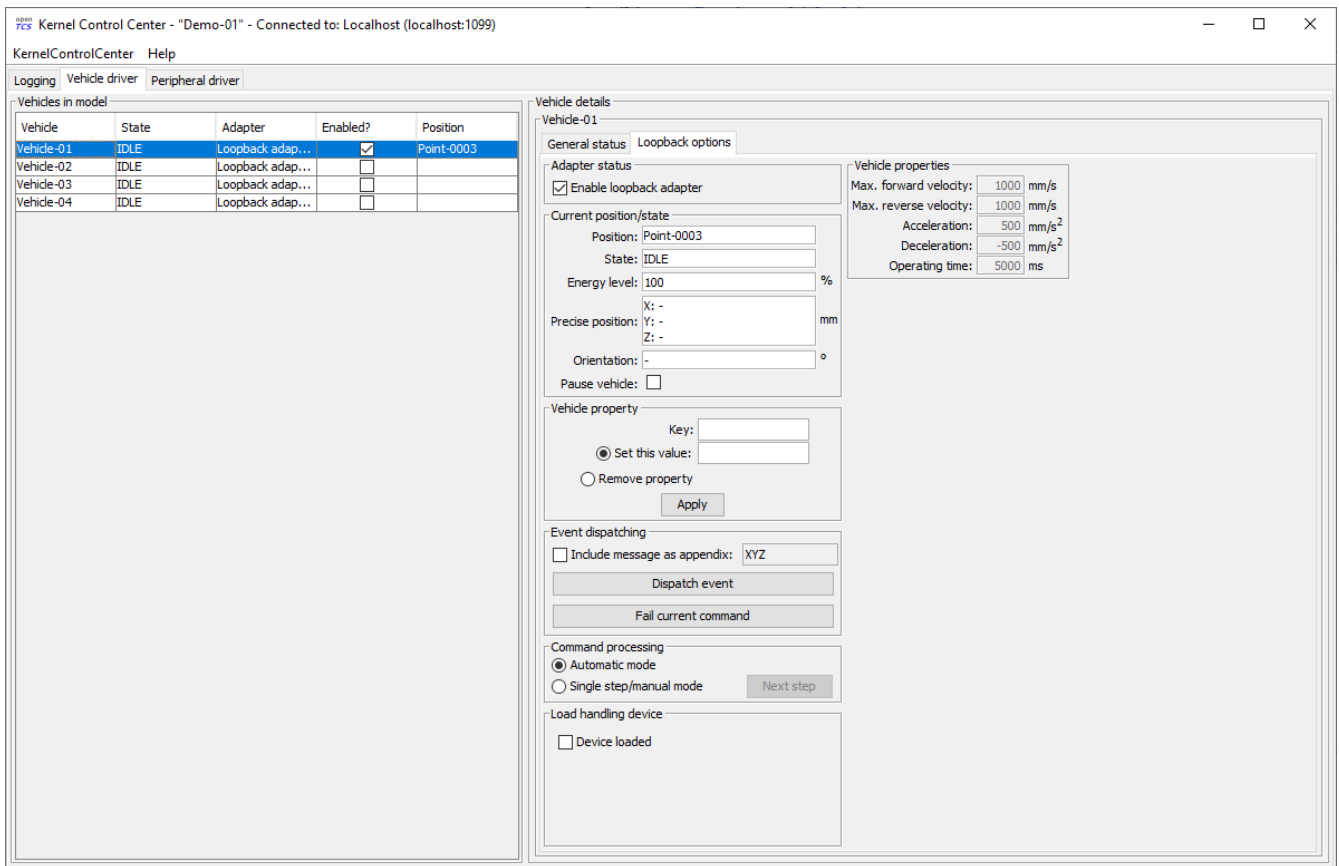
Figure 8. Driver panel with detailed view of a vehicle

3.2.2. Configuring vehicle drivers

1. Switch to the Kernel Control Center application.

2. Associate a vehicle with the loopback driver by right-clicking on the vehicle in the vehicle list of the driver panel and selecting the menu entry **[Driver › Loopback adapter (virtual vehicle)]**.

3. Open the detailed view of the vehicle by double-clicking on the vehicle's name in the list.

4. In the detailed view of the vehicle that is now shown to the right of the vehicle list, select the **[Loopback options]** tab.

5. Enable the driver by ticking the checkbox **[Enable loopback adapter]** in the **[Loopback options]** tab or the checkbox in the **[Enabled?]** column of the vehicle list.

6. In the **[Loopback options]** tab or in the vehicles list, select a point from the plant model to have the loopback adapter report this point to the kernel as the (virtual) vehicle's current position. In the **[Loopback options]** tab, this can be done by clicking on the **[Position]** text field. (In a real-world application, a vehicle driver communicating with a real vehicle would automatically report the vehicle's current position to the kernel as soon as it is known.)

7. Switch to the Operations Desk client. An icon representing the vehicle should now be shown at the point on which you placed it using the loopback driver.

8. Right-click on the vehicle and select **[Context menu › Change integration level › ...to utilize this vehicle for transport orders]** to allow the kernel to dispatch the vehicle. The vehicle is then available for processing orders, which is indicated by an integration level `TO_BE_UTILIZED` in the property panel at the bottom left of the Operations Desk application's window. (You can revert this by right-clicking on the vehicle and selecting **[Context menu › Change integration**

level › ...to respect this vehicle's position] in the context menu. The integration level shown is now `TO_BE_RESPECTED` and the vehicle will not be dispatched for transport orders any more.)

3.2.3. Creating a transport order

To create a transport order, the Operations Desk application provides a dialog window presented when selecting **[Actions › Create transport order...]** from the menu. Transport orders are defined as a sequence of destination locations at which operations are to be performed by the vehicle processing the order. You can select a destination location and operation from a dropdown menu. You may also optionally select the vehicle intended to process this order. If none is explicitly selected, the control system automatically assigns the order to a vehicle according to its internal, configurable strategies (see Default dispatcher configuration entries). You may also optionally select or define a type for the transport order to be created. Furthermore, a transport order can be given a deadline specifying the point of time at which the order should be finished at the latest. This deadline will primarily be considered when there are multiple transport orders in the pool and openTCS needs to decide which to assign next.

To create a new transport order, do the following:

1. Select the menu entry **[Actions › Create transport order...]**.

2. In the dialog shown, click the **[Add]** button and select a location as the destination and an operation which the vehicle should perform there. You can add an arbitrary number of destinations to the order this way. They will be processed in the given order.

3. After creating the transport order with the given destinations by clicking **[OK]**, the kernel will look for a vehicle that can process the order. If a vehicle is found, it is assigned the order immediately and the route computed for it will be highlighted in the Operations Desk application. The loopback driver then simulates the vehicle's movement to the destinations and the execution of the operations.

3.2.4. Withdrawing transport orders using the Operations Desk application

A transport order can be withdrawn from a vehicle that is currently processing it. When withdrawing a transport order, its processing will be cancelled and the vehicle (driver) will not receive any further movement commands for it. A withdrawal can be issued by right-clicking on the respective vehicle in the Operations Desk application, selecting **[Context menu › Withdraw transport order]** and then selecting one of the following actions:

- '...and let the vehicle finish movement': The vehicle will process any movement commands it has already received and will stop after processing them. This type of withdrawal is what should normally be used for withdrawing a transport order from a vehicle.

- '...and stop the vehicle immediately': In addition to what happens in the case of a regular withdrawal, the vehicle is also asked to discard all movement commands it has already received. (This *should* make it come to a halt very soon in most cases. However, if and how far exactly it will still move highly depends on the vehicle's type, its current situation and how communication between openTCS and this type of vehicle works.) Furthermore, all reservations for resources on the withdrawn route (i.e. the next paths and points) except for the vehicle's currently reported position are cancelled, making these resources available to other vehicles.

This forced withdrawal should be used with great care and usually only when the vehicle is currently *not moving*!

Since forced withdrawal frees paths and points previously reserved for the vehicle, it is possible that other vehicles acquire and use these resources themselves right after the withdrawal. At the same time, if the vehicle was moving when the withdrawal was issued, it may - depending on its type - not have come to a halt, yet, and still move along the route it had previously been ordered to follow. As the latter movement is not coordinated by openTCS, this can result in a *collision or deadlock* between the vehicles! For this reason, it is highly recommended to issue a forced withdrawal only if it is required for some reason, and only if the vehicle has already come to a halt on a position in the driving course or if other vehicles need not be taken into account. In all other cases, the regular withdrawal should be used.

Processing of a withdrawn transport order *cannot* be resumed later. To resume a transportation process that was interrupted by withdrawing a transport order, you need to create a new transport order, which may, of course, contain the same destinations as the withdrawn one. Note, however, that the new transport order may not be created with the same name. The reason for this is:

1. Names of transport orders need to be unique.

2. Withdrawing a transport order only aborts its processing, but does not remove it from the kernel's memory, yet. The transport order data is kept as historical information for a while before it is completely removed. (For how long the old order is kept depends on the kernel application's configuration — see Order pool configuration entries.)

As a result, a name used for a transport order may eventually be reused, but only after the actual data of the old order has been removed.

3.2.5. Continuous creation of transport orders

The Operations Desk application can easily be extended via custom plugins. As a reference, a simple load generator plugin is included which here also serves as a demonstration of how the system looks like during operation. Details about how custom plugins can be created and integrated into the Operations Desk application can be found in the developer's guide.

1. In the Operations Desk application, select **[View › Plugins › Continuous load]** from the menu.

2. Choose a trigger for creating new transport orders: New orders will either be created only once, or whenever the number of unprocessed orders in the system drops below a specified limit, or after a specified timeout has expired.

3. By using an order profile you may decide whether the transport orders' destinations should be chosen randomly or whether you want to choose them yourself.

   Using **[Create orders randomly]**, you define the number of transport orders that are to be generated at a time, and the number of destinations a single transport order should contain. Since the destinations will be selected randomly, the orders created do not necessarily make

sense for a real-world system.

Using **[Create orders according to definition]**, you can define an arbitrary number of transport orders, each with an arbitrary number of destinations and properties, and save and load your list of transport orders.

4. Start the order generator by activating the corresponding checkbox at the bottom of the **[Continuous load]** panel. The load generator will then generate transport orders according to its configuration until the checkbox is deactivated or the panel is closed.

3.2.6. Configuring peripheral device drivers

In order to perform the following steps, make sure you first associate a location with a peripheral device, as described in Adding and configuring peripheral devices.

1. Switch to the Kernel Control Center application.

2. Select the **[Peripheral driver]** tab.

   ◦ *A location should already be associated with the peripheral loopback driver.*

3. Open the detailed view of the location by double-clicking on the location's name in the list.

4. In the detailed view of the location that is now shown to the right of the peripheral device list, select the **[Loopback options]** tab.

5. Enable the driver by ticking the checkbox in the **[Enabled?]** column of the peripheral device list.

6. Switch to the Operations Desk client.

7. The peripheral device is now available for processing peripheral jobs.

3.2.7. Creating a peripheral job

A peripheral job is defined as a single operation that is to be performed by the peripheral device processing the job. Peripheral jobs can be created either explicitly or implicitly; both ways are described in the following sections.

For information on how peripheral jobs are assigned to peripheral devices, see Default peripheral job dispatcher. For information on how the control system's internal strategies for assigning peripheral jobs can be configured, see Default peripheral job dispatcher configuration entries.

3.2.7.1. Explicit creation of peripheral jobs

To create a new peripheral job, do the following in the Operations Desk application:

1. Select the menu entry **[Actions › Create peripheral job…]**.

2. In the dialog shown, select the location associated with the peripheral device that should be assigned with the peripheral job and an operation which the peripheral device should perform.

Additionally, enter some arbitrary text for the reservation token. (For the moment, the reservation token doesn't really matter. The only important thing is that it is not empty. For more details on the reservation token, see Default peripheral job dispatcher.)

3. After creating the peripheral job by clicking **[OK]**, the kernel will try to assign it to the peripheral device associated with the given location to process the job. Once the peripheral job is assigned, the loopback driver simulates the peripheral devices's execution of the operation.

> While it is fine to use this option for creating peripheral jobs, the next option is the preferred one as it allows direct interaction between vehicles and peripheral devices.

3.2.7.2. Implicit creation of peripheral jobs

> In order for implicit creation of peripheral jobs to work, make sure to first define peripheral operations on paths as described in Adding and configuring peripheral devices.

Peripheral jobs can be created implicitly by vehicles as they traverse paths that have peripheral operations defined on them. When exactly a peripheral job is created for a peripheral operation defined on a path depends on the configuration of the peripheral operation itself. As described in Peripheral operation, the *execution trigger* of a peripheral operation defines when the operation is to be performed by the peripheral device. Note the following regarding the creation time of the peripheral job belonging to a peripheral operation:

- `AFTER_ALLOCATION` (previously named `BEFORE_MOVEMENT`): The peripheral job is created as soon as the resource for the corresponding path has been allocated for the vehicle. This means that if a vehicle can accept multiple (points and) paths in advance, a peripheral job may be created before the vehicle even reaches the actual path.

- `AFTER_MOVEMENT`: The peripheral job is created as soon as the vehicle has completely traversed the path and reported the corresponding movement command as executed (i.e. when the vehicle reached the respective end point of the path).

3.2.8. Withdrawing peripheral jobs using the Operations Desk application

How a peripheral job can be withdrawn depends on whether it is related to a transport order:

- Peripheral jobs that are not related to transport orders and peripheral jobs that are related to transport orders in a final state (finished or failed) can be withdrawn by selecting them in the peripheral jobs table and clicking the withdrawal button shown above the table.

- Peripheral jobs that are related to transport orders will implicitly be aborted when the respective transport order is withdrawn. Withdrawing them independently from the related transport order is not supported (unless the related transport order is already in a final state — see above.)

  ◦ For transport orders that are withdrawn regularly, related peripheral jobs are aborted after the vehicle has completed its pending movements.

  ◦ For transport orders that are withdrawn forcibly, related peripheral jobs are aborted

immediately.

3.2.8.1. Consequence of peripheral jobs failing

When a peripheral job fails, the related transport order (if any) is implicitly withdrawn under the following circumstances:

- The peripheral job was created implicitly by a vehicle traversing a path (see Implicit creation of peripheral jobs).
- The *completion required* flag of the peripheral job's peripheral operation is set to `true`.

In such cases, it is not reasonable to continue the transport order, as the *completion required* flag not only indicates that the vehicle should wait for the peripheral operation to be completed, but also for it to be completed *successfully*. Otherwise, the prerequisites for continuing driving / the transport order may not be met.

3.2.9. Removing a vehicle from a running system

There may be situations in which you want to remove a single vehicle from a system, e.g. because the vehicle temporarily cannot be used by openTCS due to a hardware defect that has to be dealt with first. The following steps will ensure that no further transport orders are assigned to the vehicle and that the resources it might still be occupying are freed for use by other vehicles.

1. In the Operations Desk application, right-click on the vehicle and select **[Context menu › Change integration level › ...to ignore this vehicle]** to disable the vehicle for transport order processing and to free the point in the driving course that the vehicle is occupying.
2. In the Kernel Control Center application, disable the vehicle's driver by unticking the checkbox **[Enable loopback adapter]** in the **[Loopback options]** tab or the checkbox in the **[Enabled?]** column of the vehicle list.

3.2.10. Pausing and resuming the operation of vehicles

The Operations Desk application provides two buttons with which the operation of vehicles can be paused or resumed. However, in order for these buttons to have any effect, the respective vehicle drivers for the vehicles in use have to implement and support this feature.
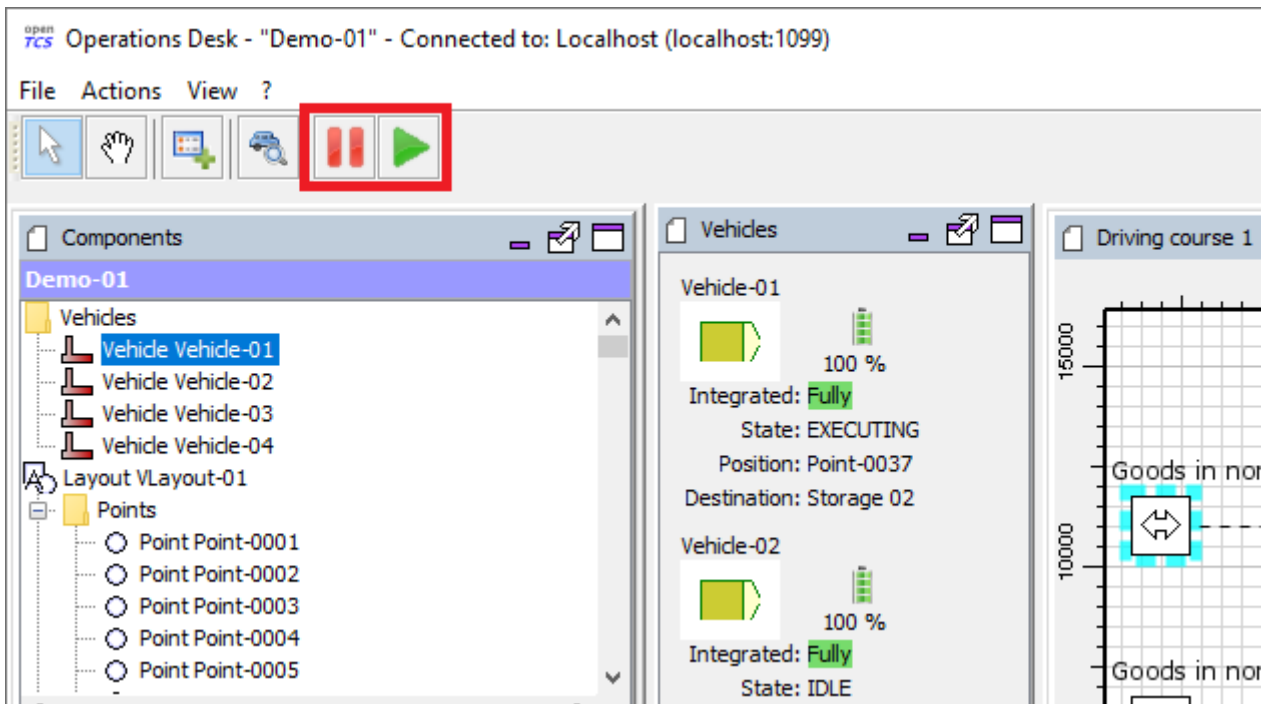
Figure 9. Pause and resume buttons in the Operations Desk application

Chapter 4. Default strategies

openTCS comes with a default implementation for each of the strategy modules. These implementations can easily be replaced to adapt to project-specific requirements. (See developer's guide.)

4.1. Default dispatcher

When either a transport order or a vehicle becomes available, the dispatcher needs to decide what should happen with which transport order and which vehicle should do what. To make this decision, the default dispatcher takes the following steps:

1. New transport orders are prepared for processing. This includes checking general routability and unfinished dependencies.

2. Updates of processes that are currently active are performed. This includes:

   ◦ Withdrawals of transport orders

   ◦ Successful completion of transport orders

   ◦ Assignment of subsequent transport orders for vehicles that are processing order sequences

3. Vehicles that are currently unoccupied are assigned to processable transport orders, if possible.

   ◦ Criteria for a vehicle to be taken into account are:

     ▪ It must be at a known position in the driving course.

     ▪ It may not be assigned to a transport order, or the assigned transport order must be *dispensable*. That is the case with parking orders, for instance, or with recharging orders if the vehicle's energy level is not critical.

     ▪ Its energy level must not be critical.

   ◦ Criteria for a transport order to be taken into account are:

     ▪ It must be generally dispatchable.

     ▪ It must not be part of an order sequence that is already being processed by a vehicle.

   ◦ The assignment mechanics are as following:

     ▪ If there are less unoccupied vehicles than processable transport orders, the list of vehicles is sorted by configurable criteria. The default dispatcher then iterates over the sorted list and, for every vehicle, finds all orders processable by it, computes the required routes, sorts the candidates by configurable criteria and assigns the first one.

     ▪ If there are less processable transport orders than unocuppied vehicles, the list of transport orders is sorted by configurable criteria. The default dispatcher then iterates over the sorted list and, for every transport order, finds all vehicles that could process it, computes the required routes, sorts the candidates by configurable criteria and assigns the first one.

     ▪ For configuration options regarding the sorting criteria, see Default dispatcher configuration entries.

4. Vehicles that are still unoccupied are sent to a recharging location, if possible.

    ◦ Criteria for a vehicle to be taken into account are:

        ▪ It must be at a known position in the driving course.

        ▪ Its energy level is *degraded.*

5. Vehicles that are still unoccupied are sent to a parking position, if possible.

    ◦ Criteria for a vehicle to be taken into account are:

        ▪ It must be at a known position in the driving course.

        ▪ It must not be at a parking position already.

4.1.1. Default parking position selection

When sending a vehicle to a parking position, the closest (according to the router) unoccupied position is selected by default. It is possible to assign fixed positions to vehicles instead, by setting properties with the following keys on them:

- `tcs:preferredParkingPosition`: Expected to be the name of a point in the model. If this point is already occupied, the closest unoccupied parking position (if any) is selected instead.

- `tcs:assignedParkingPosition`: Expected to be the name of a point in the model. If this point is already occupied, the vehicle is not sent to any other parking position, i.e. remains where it is. Takes precedence over `tcs:preferredParkingPosition`.

4.1.2. Optional parking position priorities

Optionally (see Default dispatcher configuration entries for how to enable it), parking positions may be explicitly prioritized, and vehicles can be reparked in a kind of "parking position queues". This can be desirable e.g. to park vehicles close to locations that are frequent first destinations for transport orders. (For example, imagine a plant in which goods are transported from A to B all the time. Even if there currently aren't any transport orders, it might nevertheless be a good idea to prefer parking positions near A to reduce reaction times when transport orders arrive.)

To assign a priority to a parking position, set a property with the key `tcs:parkingPositionPriority` on the point. The property's value should be a decimal integer, with lower values resulting in a higher priority for the parking position.

4.1.3. Default recharging location selection

When sending a vehicle to a recharge location, the closest (according to the router) unoccupied position is selected by default. It is possible to assign fixed positions to vehicles instead, by setting properties with the following keys on them:

- `tcs:preferredRechargeLocation`: Expected to be the name of a location. If this location is already occupied, the closest unoccupied recharging location (if any) is selected instead.

- `tcs:assignedRechargeLocation`: Expected to be the name of a location. If this location is already occupied, the vehicle is not sent to any other recharging location. Takes precedence over `tcs:preferredRechargeLocation`.

4.1.4. Immediate transport order assignment

In addition to the *implicit* assignment of transport orders according to the flow and rules described in the previous sections, transport orders can also be assigned *explicitly* (i.e. immediately). Immediate assignment of transport orders is supported for transport orders that have their intended vehicle set. This can be helpful in situations where a transport order and its intended vehicle are generally in a state where an assignment would be possible, but is prevented by certain filter criteria in the regular dispatcher flow.

Although the immediate assignment of transport orders bypasses some of the filter criteria in the regular dispatcher flow, it works only in specific situations. Regarding the transport order's state:

- The transport order's state must be `DISPATCHABLE`.
- The transport order must not be part of an order sequence.
- The transport order's intended vehicle must be set.

As for the (intended) vehicle's state:

- The vehicle's processing state must be `IDLE`.
- The vehicle's state must be `IDLE` or `CHARGING`.
- The vehicle's integration level must be `TO_BE_UTILIZED`.
- The vehicle must be reported at a known position.
- The vehicle must not process an order sequence.

> In addition to the state of the respective transport order and its intended vehicle, the dispatcher may have further implementation-specific reasons to reject an immediate assignment.

4.2. Default router

The default router finds the cheapest route from one position in the driving course to another one. (It uses an implementation of Dijkstra's algorithm to do that.) It takes into account paths that have been locked, but not positions and/or assumed future behaviour of other vehicles. As a result, it does not route around slower or stopped vehicles blocking the way.

4.2.1. Cost functions

The cost function used for evaluating the paths in the driving course can be selected via configuration. (See Default router configuration entries, the relevant configuration entry is `defaultrouter.shortestpath.edgeEvaluators`.) The following cost functions/configuration options are available:

- `DISTANCE` (default): Routing costs are equal to the paths' lengths.
- `TRAVELTIME`: Routing costs are computed as the expected time to travel on the paths (in seconds), i.e. as path length divided by maximum allowed vehicle speed.

- `EXPLICIT_PROPERTIES`: Routing costs for a vehicle on a path are taken from path properties with keys `tcs:routingCostForward<GROUP>` and `tcs:routingCostReverse<GROUP>`. The `<GROUP>` to be used is the vehicle's routing group (see Routing groups). As an example, if a vehicle's routing group is set to "Example", routing costs for this vehicle would be taken from path properties with keys `tcs:routingCostForwardExample` and `tcs:routingCostReverseExample`. This way, different routing costs can be assigned to a path, e.g. for different types of vehicles.

  Note that, for this cost function to work properly, the values of the routing cost properties should be decimal integers.

- `HOPS`: The routing costs for every path in the model is 1, which results in the route with the least paths/points being chosen.

Developers can integrate additional custom cost functions using the openTCS API.

More than one cost function can be selected in the configuration by listing them separated by commas. The costs computed by the respective functions are then added up. For example, when using `"DISTANCE, TRAVELTIME"`, costs for routes are computed as the sum of the paths' lengths and the time a vehicle needs to pass it.

> Adding distances to durations obviously does not make sense. It is the user's responsibility to choose a configuration that is usable and appropriate for the respective use case.

4.2.2. Routing groups

It is possible to treat vehicles in a plant differently when computing their routes. This may be desirable if they have different characteristics and actually have different optimal routes through the driving course. For this to work, the paths in the model or the cost function used need to reflect this difference. This isn't done by default — the default router computes routes for all vehicles the same way unless told otherwise. To let the router know that it should compute routes for a vehicle separately, set a property with the key `tcs:routingGroup` to an arbitrary string. (Vehicles that have the same value set share the same routing table, and the empty string is the default value for all vehicles.)

4.3. Default scheduler

The default scheduler implements a simple strategy for traffic management. It does this by allowing only mutually exclusive use of resources in the plant model (points and paths, primarily), as described below.

4.3.1. Allocating resources

When an allocation of a set of resources for a vehicle is requested, the scheduler performs the following checks to determine whether the allocation can be granted immediately:

1. Check if the requested resources are generally available for the vehicle.

2. Check if the requested resources are part of a block with the type `SINGLE_VEHICLE_ONLY`. If not, skip this check. If yes, expand the requested resource set to the effective resource set and check

if the expanded resources are available for the vehicle.

3. Check if the requested resources are part of a block with the type `SAME_DIRECTION_ONLY`. If not, skip this check. If yes, check if the direction in which the vehicle intends to traverse the block is the same the block is already being traversed by other vehicles.

If all checks succeed, the allocation is made. If any of the checks fail, the allocation is queued for later.

4.3.2. Freeing resources

Whenever resources are freed (e.g. when a vehicle has finished its movement to the next point and the vehicle driver reports this to the kernel), the allocations waiting in the queue are checked (in the order the requests happened). Any allocations that can now be made are made. Allocations that cannot be made are kept waiting.

4.3.3. Fairness of scheduling

This strategy ensures that resources are used when they are available. It does not, however, strictly ensure fairness/avoid starvation: Vehicles waiting for allocation of a large resource set may theoretically wait forever if other vehicles can keep allocating subsets of those resources continuously. Such situations are likely a hint at problems in the plant model graph's topology, which is why this deficiency is considered acceptable for the default implementation.

4.4. Default peripheral job dispatcher

When either a peripheral job or a peripheral device becomes available, the peripheral job dispatcher needs to decide what should happen with which peripheral job and which peripheral device should do what. To make this decision, the default peripheral job dispatcher takes the following steps:

1. Peripheral devices that are currently unoccupied but have their reservation token set are assigned to processable peripheral jobs, if possible.
   - Criteria for a peripheral device to be taken into account are:
     - It must not be assigned to a peripheral job.
     - It must have its reservation token set.
   - Criteria for a periphreal job to be taken into account are:
     - It must match the reservation token of a peripheral device.
     - It must be processable by a peripheral device.
   - If there are multiple peripheral jobs that meet these criteria, the oldest one according to the creation time is assigned first.
2. Peripheral devices that could not be assigned to a peripheral job with a matching reservation token have their reservation released.
   - The release of reserved peripheral devices is performed via a replaceable strategy. The default strategy releases peripheral devices according to the following rules:

- A peripheral devices's state must be `IDLE`.

- A peripheral devices's processing state must be `IDLE`.

- A peripheral devices's reservation token must be set.

3. Peripheral devices that are currently unoccupied and do not have their reservation token set are assigned to processable peripheral jobs, if possible.

  - Criteria for a peripheral device to be taken into account are:

    - It must not be assigned to a peripheral job.

    - It must not have its reservation token set.

  - Cirteria for a peripheral job to be taken into account are:

    - It must be generally available to be processed by a peripheral device.

    - It must be processable by a peripheral device.

  - The selection of a peripheral job for a peripheral device is performed via a replaceable strategy. The default strategy selects peripheral jobs according to the following rules:

    - The location of a peripheral job's operation must match the given location.

    - If there are multiple peripheral jobs that meet these criteria, the oldest one according to the creation time is selected.

4.4.1. Reservation token

As described above, reservation tokens are relevant for the assignment of peripheral jobs to peripheral devices. This section describes the different types of reservation tokens:

1. Reservation tokens for transport orders.

  - Optionally, transport orders can be provided with a reservation token.

  - If a transport order's reservation token is set, it is used for peripheral jobs that are created in the context of the transport order (i.e., for peripheral jobs that are created implicitly by vehicles processing a transport order - see Implicit creation of peripheral jobs).

2. Reservation tokens for peripheral jobs.

  - Peripheral jobs must always be provided with a reservation token.

  - For peripheral jobs that are created implicitly by vehicles as they traverse paths that have peripheral operations defined on them, the reservation token is set to

    - the reservation token of the transport order the corresponding vehicle is processing

    - or the name of the vehicle, if the reservation token on the transport order is not set.

3. Reservation tokens for locations that represent peripheral devices.

  - Initially, the reservation token of a location representing a peripheral device is not set. This indicates that the peripheral device is generally available to accept a peripheral job with any reservation token.

  - Once the peripheral device is assigned a peripheral job, the location's reservation token is set to the peripheral job's reservation token. As a result, the peripheral device is only available for peripheral jobs with the same reservation token until the peripheral device's

reservation is released (i.e., until the peripheral device's reservation token is reset).

Chapter 5. Configuring openTCS

5.1. Application language

By default, all openTCS applications with user interfaces display texts in English language. The applications are prepared for internationalization, though, and can be configured to display texts in a different language, provided there is a translation for it. The openTCS distribution comes with the default (English) language and a German translation. Additional translations can be integrated — how this is done is described in the Developer's Guide.

For setting the language, each application has a configuration entry that needs to be set to a *language tag* for the language to use. (See Kernel Control Center application configuration entries, Model Editor application configuration entries and Operations Desk application configuration entries.) Examples for language tags are:

- "en" for English

- "de" for German

- "no" for Norwegian

- "zh" for Chinese

By default, the configuration entries are set to "en", resulting in English texts. Since a German translation is included, you can switch e.g. the Operations Desk application to German by setting its `locale` configuration entry to "de". (Note that the application needs to be restarted for this.)

Configuring an application to use a language for which there is no translation will result in the default (English) language to be used.

5.2. Kernel configuration

The kernel application reads its configuration data from the following files:

1. `config/opentcs-kernel-defaults-baseline.properties`,

2. `config/opentcs-kernel-defaults-custom.properties` and

3. `config/opentcs-kernel.properties`.

The files are read in this order, and configuration values set in one file can be overridden in any subsequent one. For users, it is recommended to leave the first two files untouched and set overriding values and project-specific configuration data in `opentcs-kernel.properties` only.

5.2.1. Kernel application configuration entries

The kernel application itself can be configured using the following configuration entries:

`kernelapp.autoEnableDriversOnStartup`
- Type: Boolean

- Trigger for changes to be applied: when/after plant model is loaded

- Description: Whether to automatically enable drivers on startup.

kernelapp.autoEnablePeripheralDriversOnStartup

- Type: Boolean

- Trigger for changes to be applied: when/after plant model is loaded

- Description: Whether to automatically enable peripheral drivers on startup.

kernelapp.saveModelOnTerminateModelling

- Type: Boolean

- Trigger for changes to be applied: when/after plant model is loaded

- Description: Whether to implicitly save the model when leaving modelling state.

kernelapp.saveModelOnTerminateOperating

- Type: Boolean

- Trigger for changes to be applied: when/after plant model is loaded

- Description: Whether to implicitly save the model when leaving operating state.

kernelapp.updateRoutingTopologyOnPathLockChange

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to implicitly update the router's topology when a path is (un)locked.

kernelapp.rerouteOnRoutingTopologyUpdate

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether vehicles should be rerouted immediately on topology changes.

kernelapp.rerouteOnDriveOrderFinished

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether vehicles should be rerouted as soon as they finish a drive order.

5.2.2. Order pool configuration entries

The kernel's transport order pool can be configured using the following configuration entries:

orderpool.sweepAge

- Type: Integer

- Trigger for changes to be applied: instantly

- Description: The minimum age of orders or peripheral jobs to remove in a sweep (in ms).

`orderpool.sweepInterval`

- Type: Long

- Trigger for changes to be applied: when/after plant model is loaded

- Description: The interval between sweeps (in ms).

5.2.3. Default dispatcher configuration entries

The default dispatcher can be configured using the following configuration entries:

`defaultdispatcher.orderCandidatePriorities`

- Type: Comma-separated list of strings

- Trigger for changes to be applied: on application start

- Description: Keys by which to prioritize transport order candidates for assignment.
  Possible values:
  BY_AGE: Sort by transport order age, oldest first.
  BY_DEADLINE: Sort by transport order deadline, most urgent first.
  DEADLINE_AT_RISK_FIRST: Sort orders with deadlines at risk first.
  BY_COMPLETE_ROUTING_COSTS: Sort by complete routing costs, lowest first.
  BY_INITIAL_ROUTING_COSTS: Sort by routing costs for the first destination.
  BY_ORDER_NAME: Sort by transport order name, lexicographically.

`defaultdispatcher.orderPriorities`

- Type: Comma-separated list of strings

- Trigger for changes to be applied: on application start

- Description: Keys by which to prioritize transport orders for assignment.
  Possible values:
  BY_AGE: Sort by age, oldest first.
  BY_DEADLINE: Sort by deadline, most urgent first.
  DEADLINE_AT_RISK_FIRST: Sort orders with deadlines at risk first.
  BY_NAME: Sort by name, lexicographically.

`defaultdispatcher.vehicleCandidatePriorities`

- Type: Comma-separated list of strings

- Trigger for changes to be applied: on application start

- Description: Keys by which to prioritize vehicle candidates for assignment.
  Possible values:
  BY_ENERGY_LEVEL: Sort by energy level of the vehicle, highest first.
  IDLE_FIRST: Sort vehicles with state IDLE first.
  BY_COMPLETE_ROUTING_COSTS: Sort by complete routing costs, lowest first.
  BY_INITIAL_ROUTING_COSTS: Sort by routing costs for the first destination.
  BY_VEHICLE_NAME: Sort by vehicle name, lexicographically.

`defaultdispatcher.vehiclePriorities`

- Type: Comma-separated list of strings

- Trigger for changes to be applied: on application start

- Description: Keys by which to prioritize vehicles for assignment.
  Possible values:
  BY_ENERGY_LEVEL: Sort by energy level, highest first.
  IDLE_FIRST: Sort vehicles with state IDLE first.
  BY_NAME: Sort by name, lexicographically.

defaultdispatcher.deadlineAtRiskPeriod

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The time window (in ms) before its deadline in which an order becomes urgent.

defaultdispatcher.assignRedundantOrders

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether orders to the current position with no operation should be assigned.

defaultdispatcher.dismissUnroutableTransportOrders

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether unroutable incoming transport orders should be marked as UNROUTABLE.

defaultdispatcher.reroutingImpossibleStrategy

- Type: String

- Trigger for changes to be applied: instantly

- Description: The strategy to use when rerouting of a vehicle results in no route at all.
  The vehicle then continues to use the previous route in the configured way.
  Possible values:
  IGNORE_PATH_LOCKS: Stick to the previous route, ignoring path locks.
  PAUSE_IMMEDIATELY: Do not send further orders to the vehicle; wait for another rerouting opportunity.
  PAUSE_AT_PATH_LOCK: Send further orders to the vehicle only until it reaches a locked path; then wait for another rerouting opportunity.

defaultdispatcher.parkIdleVehicles

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to automatically create parking orders for idle vehicles.

defaultdispatcher.considerParkingPositionPriorities

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to consider parking position priorities when creating parking orders.

defaultdispatcher.reparkVehiclesToHigherPriorityPositions

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to repark vehicles to parking positions with higher priorities.

defaultdispatcher.rechargeIdleVehicles

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to automatically create recharge orders for idle vehicles.

defaultdispatcher.keepRechargingUntilFullyCharged

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether vehicles must be recharged until they are fully charged.
  If false, vehicle must only be recharged until sufficiently charged.

defaultdispatcher.idleVehicleRedispatchingInterval

- Type: Integer

- Trigger for changes to be applied: when/after plant model is loaded

- Description: The interval between redispatching of vehicles.

5.2.4. Default router configuration entries

The default router can be configured using the following configuration entries:

defaultrouter.routeToCurrentPosition

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to compute a route even if the vehicle is already at the destination.

The shortest path algorithm can be configured using the following configuration entries:

defaultrouter.shortestpath.algorithm

- Type: String

- Trigger for changes to be applied: on application start

- Description: The routing algorithm to be used. Valid values:
  'DIJKSTRA': Routes are computed using Dijkstra's algorithm.
  'BELLMAN_FORD': Routes are computed using the Bellman-Ford algorithm.
  'FLOYD_WARSHALL': Routes are computed using the Floyd-Warshall algorithm.

defaultrouter.shortestpath.edgeEvaluators

- Type: Comma-separated list of strings

- Trigger for changes to be applied: on application start

- Description: The types of route evaluators/cost factors to be used.
  Results of multiple evaluators are added up. Valid values:
  'DISTANCE': A route's cost equals the sum of the lengths of its paths.
  'TRAVELTIME': A route's cost equals the vehicle's expected travel time.
  'EXPLICIT_PROPERTIES': A route's cost equals the sum of the explicitly given costs extracted from path properties.
  'HOPS': A route's cost equals the number of paths it consists of.

The edge evaluator `EXPLICIT_PROPERTIES` can be configured using the following configuration entries:

`defaultrouter.edgeevaluator.explicitproperties.defaultValue`

- Type: String

- Trigger for changes to be applied: instantly

- Description: The default value used as the routing cost of an edge if no property is set on the corresponding path.
  The value should be an integer. If it is not, the edge is excluded from routing.

5.2.5. Default peripheral job dispatcher configuration entries

The default peripheral job dispatcher can be configured using the following configuration entries:

`defaultperipheraljobdispatcher.idlePeripheralRedispatchingInterval`

- Type: Integer

- Trigger for changes to be applied: when/after plant model is loaded

- Description: The interval between redispatching of peripheral devices.

5.2.6. Admin web API configuration entries

The kernel's admin web API can be configured using the following configuration entries:

`adminwebapi.enable`

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to enable the admin interface.

`adminwebapi.bindAddress`

- Type: IP address

- Trigger for changes to be applied: on application start

- Description: Address to which to bind the HTTP server, e.g. 0.0.0.0. (Default: 127.0.0.1.)

`adminwebapi.bindPort`

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: Port to which to bind the HTTP server.

5.2.7. Service web API configuration entries

The kernel's service web API can be configured using the following configuration entries:

`servicewebapi.enable`
- Type: Boolean
- Trigger for changes to be applied: on application start
- Description: Whether to enable the interface.

`servicewebapi.bindAddress`
- Type: IP address
- Trigger for changes to be applied: on application start
- Description: Address to which to bind the HTTP server, e.g. 0.0.0.0 or 127.0.0.1.

`servicewebapi.bindPort`
- Type: Integer
- Trigger for changes to be applied: on application start
- Description: Port to which to bind the HTTP server.

`servicewebapi.accessKey`
- Type: String
- Trigger for changes to be applied: instantly
- Description: Key allowing access to the API.

`servicewebapi.statusEventsCapacity`
- Type: Integer
- Trigger for changes to be applied: instantly
- Description: Maximum number of status events to be kept.

`servicewebapi.useSsl`
- Type: Boolean
- Trigger for changes to be applied: on application start
- Description: Whether to use SSL to encrypt connections.

5.2.8. RMI kernel interface configuration entries

The kernel's RMI interface can be configured using the following configuration entries:

`rmikernelinterface.enable`
- Type: Boolean
- Trigger for changes to be applied: on application start

- Description: Whether to enable the interface.

rmikernelinterface.registryPort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the RMI.

rmikernelinterface.remoteDispatcherServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote dispatcher service.

rmikernelinterface.remoteQueryServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote query service.

rmikernelinterface.useSsl

- Type: Boolean
- Trigger for changes to be applied: on application start
- Description: Whether to use SSL to encrypt connections.

rmikernelinterface.remotePeripheralServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote peripheral service.

rmikernelinterface.remotePeripheralJobServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote peripheral job service.

rmikernelinterface.remoteKernelServicePortalPort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote kernel service portal.

rmikernelinterface.remotePlantModelServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote plant model service.

rmikernelinterface.remoteTransportOrderServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote transport order service.

rmikernelinterface.remoteVehicleServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote vehicle service.

rmikernelinterface.remoteNotificationServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote notification service.

rmikernelinterface.remoteSchedulerServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote scheduler service.

rmikernelinterface.remoteRouterServicePort

- Type: Integer
- Trigger for changes to be applied: on application start
- Description: The TCP port of the remote router service.

rmikernelinterface.clientSweepInterval

- Type: Long
- Trigger for changes to be applied: on application start
- Description: The interval for cleaning out inactive clients (in ms).

5.2.9. SSL server-side encryption configuration entries

The kernel's SSL encryption can be configured using the following configuration entries:

ssl.keystoreFile

- Type: String
- Trigger for changes to be applied: on application start
- Description: The file url of the keystore.

ssl.keystorePassword

- Type: String
- Trigger for changes to be applied: on application start

- Description: The password for the keystore.

ssl.truststoreFile

- Type: String

- Trigger for changes to be applied: on application start

- Description: The file url of the truststore.

ssl.truststorePassword

- Type: String

- Trigger for changes to be applied: on application start

- Description: The password for the truststore.

5.2.10. Virtual vehicle configuration entries

The virtual vehicle (loopback communication adapter) can be configured using the following configuration entries:

virtualvehicle.enable

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to enable to register/enable the loopback driver.

virtualvehicle.commandQueueCapacity

- Type: Integer

- Trigger for changes to be applied: when/after plant model is loaded

- Description: The adapter's command queue capacity.

virtualvehicle.rechargeOperation

- Type: String

- Trigger for changes to be applied: when/after plant model is loaded

- Description: The string to be treated as a recharge operation.

virtualvehicle.rechargePercentagePerSecond

- Type: Double

- Trigger for changes to be applied: instantly

- Description: The rate at which the vehicle recharges in percent per second.

virtualvehicle.simulationTimeFactor

- Type: Double

- Trigger for changes to be applied: instantly

- Description: The simulation time factor.
  1.0 is real time, greater values speed up simulation.

virtualvehicle.vehicleLengthLoaded

- Type: Integer

- Trigger for changes to be applied: instantly

- Description: The virtual vehicle's length in mm when it's loaded.

virtualvehicle.vehicleLengthUnloaded

- Type: Integer

- Trigger for changes to be applied: instantly

- Description: The virtual vehicle's length in mm when it's unloaded.

5.2.11. Virtual peripheral configuration entries

The virtual peripheral (peripheral loopback communication adapter) can be configured using the following configuration entries:

virtualperipheral.enable

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to enable to register/enable the peripheral loopback driver.

5.3. Kernel Control Center configuration

The kernel control center application reads its configuration data from the following files:

1. `config/opentcs-kernelcontrolcenter-defaults-baseline.properties`,

2. `config/opentcs-kernelcontrolcenter-defaults-custom.properties` and

3. `config/opentcs-kernelcontrolcenter.properties`.

The files are read in this order, and configuration values set in one file can be overridden in any subsequent one. For users, it is recommended to leave the first two files untouched and set overriding values and project-specific configuration data in `opentcs-kernelcontrolcenter.properties` only.

5.3.1. Kernel Control Center application configuration entries

The kernel control center application itself can be configured using the following configuration entries:

kernelcontrolcenter.locale

- Type: String

- Trigger for changes to be applied: on application start

- Description: The kernel control center application's locale, as a BCP 47 language tag. Examples: 'en', 'de', 'zh'

kernelcontrolcenter.connectionBookmarks

- Type: Comma-separated list of <description>\|<hostname>\|<port>

- Trigger for changes to be applied: on application start

- Description: Kernel connection bookmarks to be used.

kernelcontrolcenter.connectAutomaticallyOnStartup

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to automatically connect to the kernel on startup.
  If 'true', the first connection bookmark will be used for the initial connection attempt.
  If 'false', a dialog will be shown to enter connection parameters.

kernelcontrolcenter.loggingAreaCapacity

- Type: Integer

- Trigger for changes to be applied: instantly

- Description: The maximum number of characters in the logging text area.

kernelcontrolcenter.enablePeripheralsPanel

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to enable and show the panel for peripheral drivers.

5.3.2. SSL KCC-side application configuration entries

The kernel control center application's SSL connections can be configured using the following configuration entries:

ssl.enable

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to use SSL to encrypt RMI connections to the kernel.

ssl.truststoreFile

- Type: String

- Trigger for changes to be applied: on application start

- Description: The path to the SSL truststore.

ssl.truststorePassword

- Type: String

- Trigger for changes to be applied: on application start

- Description: The password for the SSL truststore.

5.4. Model Editor configuration

The model editor application reads its configuration data from the following files:

1. `config/opentcs-modeleditor-defaults-baseline.properties`,

2. `config/opentcs-modeleditor-defaults-custom.properties`,

3. `config/opentcs-modeleditor.properties`.

The files are read in this order, and configuration values set in one file can be overridden in any subsequent one. For users, it is recommended to leave the first two files untouched and set overriding values and project-specific configuration data in `opentcs-modeleditor.properties` only.

> Some of the following configuration entries still use the prefix 'plantoverviewapp', which is reminiscent of the old plant overview application. With openTCS 6.0, the prefix 'plantoverviewapp' will be replaced with one more suitable for the new model editor application.

5.4.1. Model Editor application configuration entries

The model editor application itself can be configured using the following configuration entries:

`plantoverviewapp.connectionBookmarks`

- Type: Comma-separated list of <description>\|<hostname>\|<port>

- Trigger for changes to be applied: on application start

- Description: Kernel connection bookmarks to be used.

`plantoverviewapp.useBookmarksWhenConnecting`

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to use the configured bookmarks when connecting to the kernel.
  If 'true', the first connection bookmark will be used for the connection attempt.
  If 'false', a dialog will be shown to enter connection parameters.

`plantoverviewapp.locale`

- Type: String

- Trigger for changes to be applied: on application start

- Description: The plant overview application's locale, as a BCP 47 language tag.
  Examples: 'en', 'de', 'zh'

`plantoverviewapp.locationThemeClass`

- Type: Class name

- Trigger for changes to be applied: on application start

- Description: The name of the class to be used for the location theme.
  Must be a class extending org.opentcs.components.plantoverview.LocationTheme

5.4.2. SSL model editor-side application configuration entries

The model editor application's SSL connections can be configured using the following configuration entries:

`ssl.enable`

- Type: Boolean
- Trigger for changes to be applied: on application start
- Description: Whether to use SSL to encrypt RMI connections to the kernel.

`ssl.truststoreFile`

- Type: String
- Trigger for changes to be applied: on application start
- Description: The path to the SSL truststore.

`ssl.truststorePassword`

- Type: String
- Trigger for changes to be applied: on application start
- Description: The password for the SSL truststore.

5.4.3. Model Editor element naming scheme configuration entries

The model editor application's element naming schemes can be configured using the following configuration entries:

`elementnamingscheme.pointPrefix`

- Type: String
- Trigger for changes to be applied: on application start
- Description: The default prefix for a new point element.

`elementnamingscheme.pointNumberPattern`

- Type: String
- Trigger for changes to be applied: on application start
- Description: The numbering pattern for a new point element.

`elementnamingscheme.pathPrefix`

- Type: String
- Trigger for changes to be applied: on application start
- Description: The default prefix for a new path element.

`elementnamingscheme.pathNumberPattern`

- Type: String
- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new path element.

elementnamingscheme.locationTypePrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new location type element.

elementnamingscheme.locationTypeNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new location type element.

elementnamingscheme.locationPrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new location element.

elementnamingscheme.locationNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new location element.

elementnamingscheme.linkPrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new link element.

elementnamingscheme.linkNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new link element.

elementnamingscheme.blockPrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new block.

elementnamingscheme.blockNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new block.

elementnamingscheme.layoutPrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new layout element.

elementnamingscheme.layoutNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new layout element.

elementnamingscheme.vehiclePrefix

- Type: String

- Trigger for changes to be applied: on application start

- Description: The default prefix for a new vehicle.

elementnamingscheme.vehicleNumberPattern

- Type: String

- Trigger for changes to be applied: on application start

- Description: The numbering pattern for a new vehicle.

5.5. Operations Desk configuration

The operations desk application reads its configuration data from the following files:

1. `config/opentcs-operationsdesk-defaults-baseline.properties`,

2. `config/opentcs-operationsdesk-defaults-custom.properties`,

3. `config/opentcs-operationsdesk.properties`.

The files are read in this order, and configuration values set in one file can be overridden in any subsequent one. For users, it is recommended to leave the first two files untouched and set overriding values and project-specific configuration data in `opentcs-operationsdesk.properties` only.

> ⓘ Some of the following configuration entries still use the prefix 'plantoverviewapp', which is reminiscent of the old plant overview application. With openTCS 6.0, the prefix 'plantoverviewapp' will be replaced with one more suitable for the new operations desk application.

5.5.1. Operations Desk application configuration entries

The operations desk application itself can be configured using the following configuration entries:

plantoverviewapp.connectionBookmarks

- Type: Comma-separated list of <description>\|<hostname>\|<port>

- Trigger for changes to be applied: on application start

- Description: Kernel connection bookmarks to be used.

plantoverviewapp.useBookmarksWhenConnecting

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether to use the configured bookmarks when connecting to the kernel.
  If 'true', the first connection bookmark will be used for the connection attempt.
  If 'false', a dialog will be shown to enter connection parameters.

plantoverviewapp.locale

- Type: String

- Trigger for changes to be applied: on application start

- Description: The plant overview application's locale, as a BCP 47 language tag.
  Examples: 'en', 'de', 'zh'

plantoverviewapp.allowForcedWithdrawal

- Type: Boolean

- Trigger for changes to be applied: instantly

- Description: Whether the forced withdrawal context menu entry should be enabled.

plantoverviewapp.frameMaximized

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether the GUI window should be maximized on startup.

plantoverviewapp.frameBoundsWidth

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The GUI window's configured width in pixels.

plantoverviewapp.frameBoundsHeight

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The GUI window's configured height in pixels.

plantoverviewapp.frameBoundsX

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The GUI window's configured x-coordinate on screen in pixels.

plantoverviewapp.frameBoundsY

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The GUI window's configured y-coordinate on screen in pixels.

plantoverviewapp.locationThemeClass

- Type: Class name

- Trigger for changes to be applied: on application start

- Description: The name of the class to be used for the location theme.
  Must be a class extending org.opentcs.components.plantoverview.LocationTheme

plantoverviewapp.vehicleThemeClass

- Type: Class name

- Trigger for changes to be applied: on application start

- Description: The name of the class to be used for the vehicle theme.
  Must be a class extending org.opentcs.components.plantoverview.VehicleTheme

plantoverviewapp.ignoreVehiclePrecisePosition

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether reported precise positions should be ignored displaying vehicles.

plantoverviewapp.ignoreVehicleOrientationAngle

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether reported orientation angles should be ignored displaying vehicles.

plantoverviewapp.userNotificationDisplayCount

- Type: Integer

- Trigger for changes to be applied: on application start

- Description: The maximum number of most recent user notifications to be displayed.

5.5.2. SSL operation desk-side application configuration entries

The operation desk application's SSL connections can be configured using the following configuration entries:

ssl.enable

- Type: Boolean

- Trigger for changes to be applied: on application start

- Description: Whether to use SSL to encrypt RMI connections to the kernel.

ssl.truststoreFile

- Type: String

- Trigger for changes to be applied: on application start

- Description: The path to the SSL truststore.

ssl.truststorePassword

- Type: String

- Trigger for changes to be applied: on application start

- Description: The password for the SSL truststore.

Chapter 6. Advanced usage examples

6.1. Configuring automatic startup

1. To automatically enable vehicle drivers on startup, set the kernel application's configuration parameter `kernelapp.autoEnableDriversOnStartup` to `true`.

6.2. Automatically selecting a specific vehicle driver on startup

Automatic attachment of vehicle drivers by default works as follows: The kernel asks every available vehicle driver if it can attach to a given vehicle and selects the first one that can. It asks the loopback driver last, as that one is always available and can attach to any vehicle, but should not prevent actual vehicle drivers to be attached. As a result, if there is only one driver for your vehicle(s), you usually do not have to do anything for it to be selected.

In some less common cases, you may have multiple vehicle drivers registered with the kernel that can all attach to the vehicles in your plant model. To automatically select a specific driver in such cases, set a property with the key `tcs:preferredAdapterClass` on the vehicles, with its value being the name of the Java class implementing the driver's adapter factory. (If you do not know this class name, ask the developer who provided the vehicle driver to you for it.)

6.3. Configuring a virtual vehicle's characteristics

The loopback driver supports some (limited) configuration of the virtual vehicle's characteristics via properties set in the plant model. You can set the properties the following way:

1. Start the Model Editor application and create or load a plant model.
2. In the Model Editor application's tree view of the plant model, select a vehicle.
3. In the table showing the vehicle's properties, click into the value field labelled **[Miscellaneous]**. In the dialog shown, add a property key and value according to the list below.
4. Save the model and upload it to the kernel as described in Saving the plant model.

The loopback driver interprets properties with the following keys:

- `loopback:initialPosition`: Set the property value to the name of a point in the plant model. When started, the loopback adapter will set the virtual vehicle's current position to this. (Default value: not set)

- `loopback:acceleration`: Set the property value to a positive integer representing an acceleration in mm/s^2. The loopback adapter will simulate vehicle movement with the given acceleration. (Default value: 500)

- `loopback:deceleration`: Set the property value to a negative integer representing an acceleration in mm/s^2. The loopback adapter will simulate vehicle movement with the given deceleration. (Default value: -500)

- `loopback:loadOperation`: Set the property value to a string representing the virtual vehicle's load operation. When the virtual vehicle executes this operation, the loopback adapter will set the its load handling device's state to *full*. (Default value: not set)

- `loopback:unloadOperation`: Set the property value to a string representing the virtual vehicle's unload operation. When the virtual vehicle executes this operation, the loopback adapter will set its load handling device's state to *empty*. (Default value: not set)

- `loopback:operatingTime`: Set the property value to a positive integer representing the virtual vehicle's operating time in milliseconds. When the virtual vehicle executes an operation, the loopback adapter will simulate an operating time accordingly. (Default value: 5000)

6.4. Running kernel and its clients on separate systems

The kernel and its clients (the Model Editor, Operations Desk and Kernel Control Center applications) communicate via the Java Remote Method Invocation (RMI) mechanism. This makes it possible to run the kernel and the clients on separate hosts, as long as a usable network connection between these systems exists.

By default, the Model Editor, the Operations Desk and the Kernel Control Center are configured to connect to a kernel running on the same host. To connect them to a kernel running on a remote host, e.g. on a host named myhost.example.com, do the following:

- For the Model Editor and the Operations Desk, set the configuration parameter `plantoverviewapp.connectionBookmarks` to `SomeDescription|myhost.example.com|1099`.

- For the Kernel Control Center, set the configuration parameter `kernelcontrolcenter.connectionBookmarks` to `SomeDescription|myhost.example.com|1099`.

The configuration value can be a comma-separated list of `<description>|<host>|<port>` sets. The applications will automatically try to connect to the first host in the list. If that fails, they will show a dialog to select an entry or enter a different address to connect to.

6.5. Encrypting communication with the kernel

By default, client applications and the kernel communicate via plain Java Remote Method Invocation (RMI) calls or HTTP requests. These communication channels can optionally be encrypted via SSL/TLS. To achieve this, do the following:

1. Generate a keystore/truststore pair (`keystore.p12` and `truststore.p12`).

   a. You can use the Unix shell script or Windows batch file (`generateKeystores.sh/.bat`) provided in the kernel application's directory for this.

   b. The scripts use the key and certificate management tool 'keytool' that is included in both the Java JDK and JRE. If 'keytool' is not contained in the system's `Path` environment variable the `KEYTOOL_PATH` variable in the respective script needs to be modified to point to the location where the 'keytool' is located.

   c. By default, the generated files are placed in the kernel application's `config` directory.

2. Copy the `truststore.p12` file to the client application's (Model Editor, Operations Desk or Kernel

Control Center) `config` directory. Leave the file in the kernel application's `config` directory as well.

3. In the kernel's configuration file, enable SSL for the RMI interface and/or for the web service interface. (See RMI kernel interface configuration entries and/or Service web API configuration entries for a description of the configuration entries.)

4. If you enabled SSL for the RMI interface, you need to enable it in the Model Editor's, Operations Desk's and the Kernel Control Center's configuration files, too. (See SSL model editor-side application configuration entries, SSL operation desk-side application configuration entries and SSL KCC-side application configuration entries for a description of the configuration entries.)

6.6. Configuring automatic parking and recharging

By default, idle vehicles remain where they are after processing their last transport order. You can change this in the kernel's configuration file:

- To order vehicles to charging locations automatically, set the configuration parameter `defaultdispatcher.rechargeIdleVehicles` to `true`. The default dispatcher will then look for locations at which the idle vehicle's recharge operation is possible and create orders to send it to such a location (if unoccupied). (Note that the string used for the operation is driver-specific.)

- To order vehicles to parking positions automatically, set the configuration parameter `defaultdispatcher.parkIdleVehicles` to `true`. The default dispatcher will then look for unoccupied parking positions and create orders to send the idle vehicle there.

6.7. Configuring order pool cleanup

By default, openTCS checks every minute for finished or failed transport orders and peripheral jobs that are older than 24 hours. These orders and jobs are removed from the pool. To customize this behaviour, do the following:

1. Set the configuration entry `orderpool.sweepInterval` to a value according to your needs. The default value is 60.000 (milliseconds, corresponding to an interval of one minute).

2. Set the configuration entry `orderpool.sweepAge` to a maximum age of finished orders and jobs according to your needs. The default value is 86.400.000 (milliseconds, corresponding to 24 hours that a finished order or job should be kept in the pool).

6.8. Using model element properties for project-specific data

Every object in the plant model - i.e. points, paths, locations, location types and vehicles - can be augmented with arbitrary project-specific data that can be used, e.g. by vehicle drivers, custom client applications, etc.. Possible uses for such data could be informing the vehicle driver about additional actions to be performed by a vehicle when moving along a path in the model (e.g. flashing direction indicators, displaying a text string on a display, giving an acoustic warning) or controlling the behaviour of peripheral systems (e.g. automatic fire protection gates).

The data can be stored in properties, i.e. key-value pairs attached to the model elements, where both the key and the corresponding value are text strings. These key-value pairs can be created and edited using the Model Editor application: Simply select the model element you want to add a key-value pair to and click into the value field labelled **[Miscellaneous]** in the properties table. In the dialog shown, set the key-value pairs you need to store your project-specific information.

> For your project-specific key-value pairs, you may specify arbitrary keys. openTCS itself will not make any use of this data; it will merely store it and provide it for custom vehicle drivers and/or other extensions. You should, however, not use any keys starting with `"tcs:"` for storing project-specific data. Any keys with this prefix are reserved for official openTCS features, and using them could lead to collisions.