

Booklet Cover Page



MPI Interface Documentation

Version 2.0 Draft-1

1 Table of Contents

1.	Table of Contents	1-2
2.	General Information	3
3.	Introduction to the Mobile-PI-Interface	4-6
3.1.	Change Log	6-10
3.2.	Service Maturity	10-11
3.3.	Authentication	11-12
3.4.	Services	12
3.4.1.	GPS Service	12-13
3.4.2.	Logical Positioning	13-15
3.4.3.	Train Binding	15-17
3.4.4.	Live Updates	17-18
3.4.5.	Stop On Demand	18-19
3.4.6.	Schedule Update	19-20
3.4.7.	On demand schedules	20-21
3.4.8.	Display Control	21
3.4.9.	Audio Control	21-22
3.4.10.	Diagnostics	22-24
3.4.11.	Driver Message	24
3.4.12.	Configuration	24-25

3.4.13.	Passenger Counting	25-26
3.5.	Transports	26
3.5.1.	WebSocket Transport	26-28
4.	DiLoc JSON Schedule Format	29
4.1.	Introduction	29-36
4.2.	Schema	36
5.	Index	37

2 General Information

The Mobile-PI-Interface specification is the intellectual property of CN-Consult GmbH.

All modifications and extensions to this interface must be coordinated and agreed with CN-Consult GmbH. Requested changes are documented by CN-Consult GmbH and made available freely to everyone interested.

The detailed developer documentation (including the xml-schema, detailed explanations and examples) can be requested from the [CN-Consult GmbH-website](#). Registered users will subsequently receive all updates on the Mobile-PI-Interface specification via Email.

The specification may be freely used by any company wishing to implement a communication interface between mobile and central passenger information systems, as long as the specification is not modified or extended without cooperation with CN-Consult GmbH. (as stated above)

3 Introduction to the Mobile-PI-Interface

The Mobile-PI-Interface provides a way for **m**obile **p**assenger **i**nformation (MPI) systems to send and receive data from a DiLoc|Motion or other servers that implement passenger information features.

All data that is sent and received through this interface is categorized into functional services. A client may only implement one service, but there are services that depend on other services so that they work correctly.

Services

Each service describes several messages that may be interchanged between client and server when the service is used by both parties.

Implemented Services

The following services are currently available:

Service	Description	Dependencies	Maturity
GPS	Provides the ability to send gps location information.	-	Mature
Logical Positioning	Provides the ability to send logical positioning information.	-	Mature
Schedule Update	Provides the ability to notify the client that updated schedules are available.	-	Mature
Train Binding	Provides the ability to bind and unbind train numbers from clients.	-	Mature
Live Updates	Provides the ability for a train to receive live updates about forecasts, connections, etc.	Train Binding Level1	Mature
Stop on	Provides the ability to send	Train Binding	Mature

Service	Description	Dependencies	Maturity
Demand	information about requested on demand halts.	Level1	
Display Control	Provides the ability to control the passenger information displays in the train.	Train Binding Level1	Complete
Audio Control	Provides the ability to control the audio devices for passenger information.	Train Binding Level1	Complete
Diagnostics	Provides the ability to send diagnostic information to the server.	-	Draft
On Demand Schedules	Provides the ability to request additional spontaneous train schedules from the server.	Schedule Update	Mature
Driver Message	Provides the ability to send messages to the driver from the server and allows to confirm received messages.	-	Mature
Configuration	Provides the ability to update the configuration of a train.	-	Draft
Passenger Counting	Provides the ability to exchange passenger counts and passenger occupancy.	-	Draft

Transports

The communication may be realized with different transports. The data-format is always the same but the transport differs slightly. Currently only the [WebSocket transport](#) is implemented. In future a direct transport via a standard TCP/IP connection and a HTTP-based transport may be realized.

Communication-Format

In MPI 1.x the communication is entirely done in XML. The **MobilePI Schema (on-line documentation)** describes the format in detail. For MPI 2.0 a JSON equivalent is also in the works and will be made available with the final version of MPI 2.0 In the future (MPI 3.0) XML will may not be used anymore.

In order to be able to authenticate the communication between server and client a hash-based [authentication](#) is used.

Version

This is **Version 2.0 Draft-1** of the Mobile-PI-Interface. For the changes in this release have a look at the [change log](#).

3.1 Change Log

This contains change information about the different releases of the specification.

Version 2.0 Draft 1

Version 2.0 is a big overhaul of the mpi specification and provides a lot of additional services and information that can be easily exchanged between the server and a train.

Changes in the [diagnostics service](#):

- Requesting log files from a device
- Requesting screenshots from devices
- Device maintenance functions like device-restarts and using a device test-mode.
- Transmitting important events happening in the vehicle

A new [Configuration Service](#) that allows to send and track configuration updates.

The new [Passenger counting](#) service allows to transmit information about passenger counts and passenger occupancy.

The trainbinding service now allows to send **vehicle information ('bindtrain Element' in the on-line documentation)** to the server and the live-updates service now specifies means of transmitting the current status of **vehicle service attributes ('serviceattributestatusupdate Element' in the on-line documentation)** to the server.

Additionally the service status in the specification is now tracked by a more understandable

maturity level and the [maturity levels and their meaning](#) have been documented.

Various other additions and improvements have been made throughout the documentation.

Version 1.4.2

This version contains no schema-related changes against the previous version. So schema-wise this version is 100% the same as the already released version 1.4.1.

The only change in this version relates to [general information](#) about the MPI specification, how it may be distributed, extended etc..

Version 1.4.1

This version fixes a critical error in the schema: the tag **nexttroutestationdistance** (**'nexttroutestationdistance Element' in the on-line documentation**) is now optional as this cannot be calculated when the train arrives at the destination station.

Version 1.4

The documentation now features a **refreshed design** and there is a new page that contains example-messages for additional reference. These examples are also packaged into the web-based documentation into the folder xml-examples. Also, the **xsd-file ('MobilePI.xsd' in the on-line documentation)** is now always packaged into the web-based documentation so it is readily available within the documentation.

The outer **message ('message Element' in the on-line documentation)**-element may now contain an **mpi-version ('mpiversion Attribute' in the on-line documentation)** which allows the client to specify the version of the mpi spec it supports.

This version specifies the following new services:

- Sending messages to the driver with the addition of the new [DriverMessage service](#).
- The new [LogicalPositioning service](#) allows to transmit logical position-data as additional enhancement to the server.

Changes to existing services:

- The **gpsdata ('gpsdata Element' in the on-line documentation)**-message was enhanced to optionally contain an accuracy value.
- The forecast-message may now contain additional **advices ('advices Element' in the on-line documentation)** per station. These can be triggered flexibly and allows the server to directly send station-specific information to the client.

Version 1.3.2

The [StopOnDemand service](#) was updated to allow the messages to flow in both directions. Note: This does not introduce new/changed messages, only the documentation is updated.

This version updates the XSD and fixes the value for the DiLoc-Json-Schedule format to be **djsf** instead of **djs**.

Version 1.3.1

This version only updates the XSD and adds new enumerations for the connection category. Added values are: **ropeway**, **funicular** and **ship**.

Version 1.3

This version brings the possibility for clients to retrieve automatic connection announcements from the server.

- Added audiourl and audioformat to the **connections ('connections Element' in the on-line documentation)**-tag
- Made audioformat attribute optional as it is only available when an audiourl is present.

To ensure backwards compatibility this will only be sent from a DiLoc|Rail-Server if the feature-level of the mpi service is set to 1.3 or greater.

Version 1.2.2

- Documented support to convert mp3-files to other sampling rates in **audiourl attribute ('lang Element' in the on-line documentation)** and **url tag ('url Element' in the on-line documentation)**.

Version 1.2.1

- Fixed a bug in the schema: The **<value ('value Element' in the on-line documentation)>**-tag from systemstatus should be an extension of xs:string.
- The [diagnostics](#) and [audiocontrol](#) services were promoted to RC as parts of them are already implemented.

Version 1.2

- The **updateforecast** ('**updateforecast Element**' in the on-line documentation)-message now also supports containing information about departure times and lateness inside the <**station** ('**station Element**' in the on-line documentation)>-tag. (This will be sent from DiLoc 2.7-Beta9 and later)
- Fixed a bug in the schema: The <**connections** ('**connections Element**' in the on-line documentation)>-tag may also be empty and contain no <connection>-tag if there are no valid connections at a station anymore.
- Fixed a typo in the **systemstatus** ('**systemstatus Element**' in the on-line documentation)-message: The attribute is now correctly named softwareversion instead of sofwareversion.

Version 1.1.1

Fixed a bug in the schema: The sequence for <lang>-tags inside the <**advice** ('**advice Element**' in the on-line documentation)>-tag was incorrectly set to a maximum of 1. This was corrected and it is "unbounded" now.

No other changes were made in this version.

Version 1.1

- Improved the [Train Binding service](#) by allowing clients to remote-bind other trains by using the server as a relay. This is a backwards-compatible change that allows certain messages to be sent in both directions and also adds some new messages. To differentiate between different levels of implementations the service is now separated into three levels: Level 1 matches with the specification Version 0.5-1.0, Level 2 and Level 3 define the new features added in 1.1
- Improved the outage information of [the Live Updates service](#) by adding ready made advice texts and audio-files to the outage information. This allows clients to easily display and announce information about the outage without the need to implement TTS or other mechanisms on the train.

Version 1.0

- Improved the [Live Updates service](#) by adding outage information to the **updateforecast** ('**updateforecast Element**' in the on-line documentation) message. This is a backwards-compatible change, so existing clients implementing the live-updates service should not be affected by this additional information.
- Documented the [DiLoc JSON Schedule Format](#).

Version 0.9

- New Services: [On Demand Schedules](#) and [Diagnostics](#).
- Improvements to the audio control service (can now also send an announcement as audio-file).

Version 0.1 - 0.5

Development of the initial version.

3.2 Service Maturity

Since the MPI specification is grouped into different functional services these services may be extended and enhanced independently from each other.

This also means that different services may have a different level of maturity. The used maturity levels and their meaning are explained here.

Preliminary

This is the lowest maturity level and is used for services where only an initial draft is available that has not yet been peer reviewed.

Draft

A draft version is a version that has been internally peer reviewed and describes a complete specification. This might still be altered without prior notice if new requirements emerge or changes need to be made.

Complete

When the specification of a service is agreed upon both parties of an implementation (client and server) the maturity level is named complete. Changes and additions to the service can still be made but must be discussed between both parties and accepted by both parties.

Mature

A service that is implemented at least once for every side (client and server) and is used in working projects is considered to be mature.

Changes to the service will only be made with backwards compatibility in mind: Only new elements and attributes are added.

3.3 Authentication

In order to be able to authenticate data that is sent via the interface each **message ('message Element' in the on-line documentation)** must have a hash-attribute. The hash must be built after some strict rules so that the receiver can verify that the data is coming from a valid sender.

Hash generation rules

The hash is generated differently based on the message direction.

Client -> Server

The hash that a client needs to send to the server is defined as follows:

Pseudo Code

```
auth = sha256("client:{partnerid}:{deviceid}")
```

Where {partnerid} is to be replaced with the partner-id the interface partner gets beforehand and {deviceid} is to be replaced with the unique device-id identifying the device that sends the message.

This hash allows the server to authenticate any messages sent from the client and makes it possible to know from which device or partner a given message is, without transmitting the ids in clear text.

Server -> Client

The auth hash that is sent to a client is defined as follows:

Pseudo Code

```
auth = sha256("server:{partnerid}:{serverid}")
```

Where {partnerid} is to be replaced with the partner-id of the interface partners. It is the same partner-id that the client uses to generate its auth key. The server-id is usually always the same as long as it is not compromised. (See Banning)

This means that the auth-key sent from the server to all clients of the same partner is normally always the same.

Banning

In order to ban compromised devices or partners the server may delete known partner-ids or device-ids. After they have been deleted on the server all messages from these devices and or partners will be dropped. This means it should be easily possible to replace the partner-id or device-id on the client side if necessary.

In the same way it should also be possible to exchange the serverid on the client side so that compromised servers may be banned.

3.4 Services

3.4.1 GPS Service

The GPS-Service enables mobile passenger information systems to transmit their current gps-location to the server.

This is necessary for DiLoc|Motion to properly locate trains and be able to create train-run-messages when the train travels and to show the correct information on stationery passenger information systems.

Maturity

The GPS-Service maturity level is currently considered to be [Mature](#).

Used Messages

The messages that are sent in the context of this service are **gpsdata ('gpsdata Element' in the on-line documentation)** and **gpsdatareply ('gpsdatareply Element' in the on-line documentation)**. For more information on these tags please consult the corresponding schema documentation.

Communication

The communication flow for the GPS-Service is mainly from client to server. The client sends

gpsdata messages in regular intervals to the server. The server processes the location information and replies with a gpsdatareply message.

Requirements

In order for the server being able to make use of the sent locations some requirements must be followed.

1. Positions must be updated frequently

For the algorithms inside wayside systems (like DiLoc|Motion) to work correctly, position-data should be sent at least every 6 seconds. This is necessary to accurately detect halts at a station or through passes at a station. To not overload the server unnecessarily, data should not be sent more often than in 3 second intervals.

In case of connection problems the client should buffer as many gpsrecords as possible and send them when the connection is available again.

2. Data must be sent chronologically

If data from the past is sent to the interface it must be sent in chronological order. This means that when the Internet-Connection (usually GPRS/3G) of the vehicle is not available for some time and the client cannot send the data and saves it to be sent later it must be sent in chronological order of the acquired position, beginning with the oldest position data.

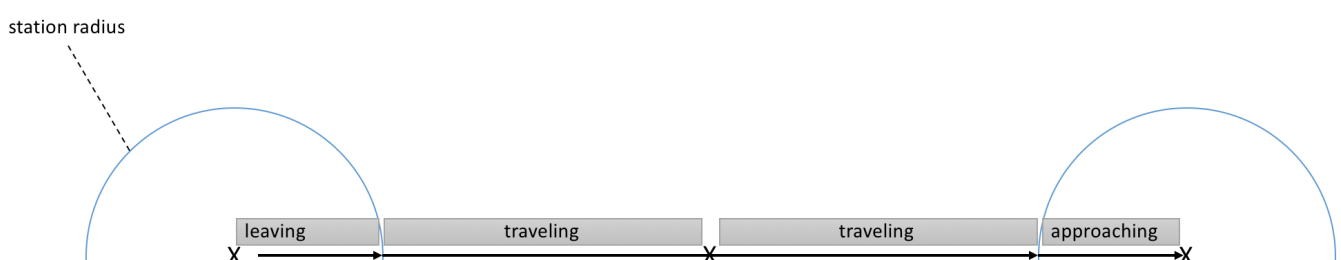
3.4.2 Logical Positioning

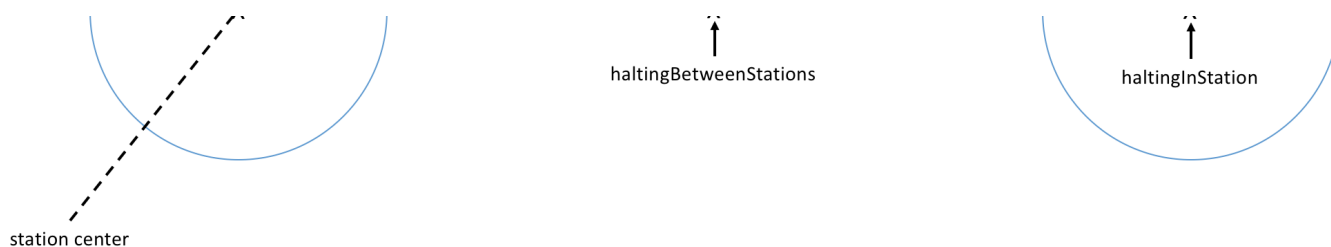
The logical positioning complements the gps positioning but is an optional service.

In contrast to the gps positioning it can also work when gps is not available, for example by using odometer information from the device.

The logical positioning records contain a reference and distance to the preceding station and a reference and distance to the next station. Additionally a status gives more information about the state the mobile device per positioning record.

The following image illustrates the state that is transmitted on a journey between two stations:





When arriving at a station, the next routestation will become the previous routestation, the distance will be set to 0 and the next routestation will be the next station on the trips route.

Most of the time it is necessary for the client to know the trip-route in advance (think forks) to be able to determine the logical position. Thus a client may only send the logical position data when bound to a train.

Maturity

The LogicalPositioning-Service maturity level is currently considered to be [Mature](#).

Used Messages

This service uses the messages **logicalpositioningdata** ('**logicalpositiondata Element**' in the **on-line documentation**) and **logicalpositioningdatareply** ('**logicalpositiondatareply Element**' in the **on-line documentation**).

Communication

The communication flow for the logical positioning service is mainly from client to server.

Requirements

In order for the server being able to make use of the sent information some requirements must be followed.

1. Positions must be updated frequently

For the algorithms on the server side to work correctly position-data should be sent at least every 10 seconds. This is necessary to be able to track devices in real time. To not overload the server unnecessarily, data should not be sent more often than in 8 second intervals.

In case of connection problems the client should buffer as many logicalpositionrecords as possible and send them when the connection is available again.

2. Data must be sent chronologically

If data from the past is sent to the interface it must be sent in chronological order. This means that when the Internet-Connection (usually GPRS/3G) of the rolling stock is not available for some time and the client cannot send the data and saves it to be sent later it must be sent in chronological order of the determined position, beginning with the oldest position data.

3.4.3 Train Binding

The train binding service enables mobile passenger information systems to bind the current train number to the device.

This is needed on the server side so that the server knows which device/client is currently operating which train. Many other services (like for example the [LiveUpdates](#) or [StopOnDemand](#)-service) depend on this service being implemented correctly by the client for their part to work correctly.

Since MPI-Interface 1.1 it is also possible that the server can send a message to the client to inform it that it should now drive a particular train number. (See Remote Train-Binding for more information about that)

Since MPI 2.0 the client may optionally also send vehicle composition information as part of the trainbind. This data is destined to be used for vehicle formation displays and for vehicle diagnostic purposes.


Maturity

The TrainBinding-Service maturity level is currently considered to be [Mature](#).

Used Messages

This service uses the messages **bindtrain** ('**bindtrain Element**' in the **on-line documentation**), **bindtrainreply** ('**bindtrainreply Element**' in the **on-line documentation**), **unbindtrain** ('**unbindtrain Element**' in the **on-line documentation**) and **unbindtrainreply** ('**unbindtrainreply Element**' in the **on-line documentation**) for the actual train-binding.

Additionally it is possible for a client to remotely bind other devices to a train number. This part of the service uses the **relaybindtrain** ('**relaybindtrain Element**' in the **on-line documentation**), **relaybindtrainstatus** ('**relaybindtrainstatus Element**' in the **on-line documentation**), **relayunbindtrain** ('**relayunbindtrain Element**' in the **on-line documentation**), **relayunbindtrainstatus** ('**relayunbindtrainstatus Element**' in the **on-line documentation**).

 **Note:** remotely binding another device to a train number is completely optional and must only be implemented if for example two vehicles drive the same train number and have no means to communicate with each other.

Communication

The communication flow for the TrainBinding-Service is mainly from client to server. The client sends information about the current binding to the server and the server replies to it.

Since MPI-Interface 1.1 it is also possible that a client may receive a bindtrain message or unbindtrain message. In this case the communication flow is reversed.

Remote Train-Binding

This provides a solution to let clients bind other clients to trains if they have no direct means of communication between each other. The server acts as a simple relay that forwards the message to the other device.

Remote Train-Binding is new as of MPI 1.1. The change is backwards compatible however: DiLoc will never send bindtrain and unbindtrain-messages on its own to old clients. However if a client sends a relaybindtrain or relayunbindtrain message then DiLoc **will** send bindtrain and unbindtrain messages to other clients. So if one client is capable of sending relaybindtrain and relayunbindtrain messages all other clients should also be able to process such messages or at least reply with a proper error message.

Remote train-binding is the only exception where it is allowed to send multiple relaybindtrain messages shortly after another without waiting that the remote bindtrain was finished. However if this is used by the client it is in the responsibility of the client to properly correlate incoming relaybindtrainstatus messages to the previously sent relaybindtrain message. (This can be done with the train number and the device identifier which should be unique for the time frame of 10 minutes)

Requirements

Since the train-binding service is a core service that many other services depend on its really important that this is implemented correctly, especially in corner cases.

The server saves the current binding of a device through reconnects and server restarts. That means if the connection is reestablished the client should have in mind that the server still has the binding information. This makes it easy to communicate with the server when the connection is reestablished because nothing special has to be sent before other messages if the binding did not change. However if the binding changed during a time where no connection to the server was available, the client should rebind as soon as possible when the connection is up again. For efficiency reasons however the client should only rebind if the binding really changed.

Implementation levels

This service may be implemented in three levels of completeness.

Level 1 (MPI 0.5)

This means the client sends bindtrain and unbindtrain messages correctly and handles bindtrainreply and unbindtrainreply messages correctly. This is the most basic level that every client should implement.

This also matches with what MPI 0.5 specified for this service. In this regard nothing changed, so the new messages are pure addons that do not break backwards compatibility. All clients that fully implemented the TrainBinding Service of Version 0.5 are now compatible with Level 1 of the service.

Level 2

In addition to Level 1 this means the client is also able to process bindtrain and unbindtrain messages and it sends correct bindtrainreply and unbindtrainreply messages.

Level 3

In addition to Level 2 this means the client is also able to send relaybindtrain and relayunbindtrain messages and processes relaybindtrainstatus and relayunbindtrainstatus messages.

3.4.4 Live Updates

The LiveUpdates-Service provides mechanisms to send up to date information to the train regarding its forecast and train connections.

Since MPI 2.0 it also allows the client to inform the server about real-time changes to the vehicles service attributes.

Maturity

The LiveUpdates-Service maturity level is currently considered to be [Mature](#).

Used Messages

The server sends the messages **updateconnections** ('**updateconnections Element**' in the on-

line documentation) and **updateforecast** (**'updateforecast Element' in the on-line documentation**). The **updateconnections**-message is used to update the connections that passengers may reach at a certain station. The **updateforecast**-message is used to send lateness and outage information to the client. This is for example useful because the server knows about latenesses in advance for example because of the need to wait for another train at a station in the future.

As of DiLoc|Motion 2.7 the forecast also contains information about an outage the train will have in the future. This is also helpful so that the passenger information system may inform the passengers in advance that they must leave the train because of construction work or other reasons. The outage information contained in the forecast can be used to generate announcements and show messages in indoor TFT-Screens or enrich the displayed pearl-chain with information about the outage. The outage also contains ready made messages in different languages and corresponding audio files that may be shown as message and played as announcements. When and how these information is displayed is up to the software in the train.

Currently all clients automatically get this data sent after a train is bound to it. For later versions this may be configurable per client or even be dynamic in a sense that the client must request receiving updates for forecasts and/or connections.

Since MPI 2.0 the messages **serviceattributeupdate** (**'serviceattributestatusupdate Element' in the on-line documentation**) and **serviceattributestatusupdatereply** (**'serviceattributestatusupdatereply Element' in the on-line documentation**) can be used to update the server about changes in the status of the services the vehicles provides. Such services may be restrooms, wheelchair compartments, a restaurant, passenger wifi or air conditioning.

Communication

The communication flow for the LiveUpdates-Services is mainly from server to client. The server sends updated data to the client so that it can update its passenger information.

Requirements

In order that the server knows which live data must be sent to which client (train) the client must implement the [TrainBinding-service](#).

3.4.5 Stop On Demand

The stop on demand service enables mobile passenger information systems to receive information about requested stops at stations that have conditional halts.

This can be useful for DiLoc|Motion to send stop requests to the HMI of the driver so that he can see that a stop is needed at a particular station. It can also be used the other way round, so

that DiLoc|Rail knows that a stop was requested from inside the train, so that this information is also available on the stationary side.

Maturity

The StopOnDemand-Service maturity level is currently considered to be [Mature](#).

Used Messages

The used messages in this service are **stoprequest** ('**stoprequest Element**' in the **on-line documentation**) and **stoprequestconfirmation** ('**stoprequestconfirmation Element**' in the **on-line documentation**).

Communication

The communication flow for the StopOnDemand-Service can be from server to client, or from client to server. One side sends a stop-request message, and the other end confirms the received-stop-request.

If the DiLoc|Motion server receives a stoprequest it makes sure that this is not bounced back as new stoprequest to the client.

Requirements

In order that the server knows which stoprequest must be sent to which client (train) the client must implement the [TrainBinding-service](#).

Stop-requests are really time sensitive. This means they should be processed as fast as possible by the client so that the time that passes between a customer presses the stop button until the stop-request is displayed in the HMI of the driver is kept as minimal as possible.

If the client receives stop-requests it must make sure to not bounce back the received stop-request as new stop-request to the server, otherwise endless communication loops can happen.

3.4.6 Schedule Update

This service provides the possibility to notify a client that new schedule data is ready to be downloaded.

Maturity

The ScheduleUpdate-Service maturity level is currently considered to be [Mature](#).

Used Messages

This service uses the messages **scheduleupdate** ('**scheduleupdate Element**' in the **on-line documentation**) and **scheduleupdatestatus** ('**scheduleupdatestatus Element**' in the **on-line documentation**).

Communication

The communication flow for this service is mainly from server to client. The server sends a scheduleupdate message to the client and the client answers with one or more scheduleupdatestatus messages.

Requirements

3.4.7 On demand schedules

This services allows the vehicle to load schedules (time-tables) on demand in the case a certain trip number was not (yet) contained in the currently used schedule update.

Maturity

The OnDemandSchedule-Service maturity level is currently considered to be [Mature](#).

Used Messages

This service uses the messages **requesttrainschedule** ('**requesttrainschedule Element**' in the **on-line documentation**) and **requesttrainschedulereply** ('**requesttrainschedulereply Element**' in the **on-line documentation**).

Communication

The communication flow for the on demand schedule service is mainly from client to server. The client requests the schedule for a certain train and the server answers if a train could be found or not. Depending on the requested format the train schedule is either directly delivered in the reply or a separate schedule update is sent to the client.

Requirements

If the client does not support the diloc schedule format the client must additionally implement

the [schedule update service](#).

3.4.8 Display Control

The DisplayControl-service provides ways for the server to switch all displays in a train dark or display additional text on the displays.

Maturity

The DisplayControl-Service maturity level is currently considered to be [Complete](#).

Used Messages

This service uses the **displaycommand** ('**displaycommand Element**' in the **on-line documentation**) and **displaycommandconfirmation** ('**displaycommandconfirmation Element**' in the **on-line documentation**) message.

Communication

The flow of communication is initiated from server to client. The server sends a displaycommand to the client and the client replies with a displaycommandconfirmation.

Requirements

In order that the server knows which live data must be sent to which client (train) the client must implement the [TrainBinding-service](#).

3.4.9 Audio Control

The AudioControl-service provides ways for the server to make ad hoc announcements or mute all audio devices on a train.

Maturity

The AudioControl-Service maturity level is currently considered to be [Complete](#).

Used Messages

This services uses the message **audiocommand** ('**audiocommand Element**' in the **on-line documentation**) and **audiocommandconfirmation** ('**audiocommandconfirmation Element**'

in the on-line documentation).

Communication

The flow of communication is initiated from server to client. The server sends an audiocommand to the client and the client replies with an audiocommandconfirmation.

Requirements

In order that the server knows which live data must be sent to which client (train) the client must implement the [TrainBinding-service](#).

3.4.10 Diagnostics

The Diagnostics-service provides ways for the train to send diagnostic information to the server. It provides different possibilities for the most common diagnostic requirements.

It is possible to transmit generic and expandable system status information in a flexible key/value system.

Detailed information (like general hardware information, status and capabilities) about any devices (e.g. manufacturer, serial number, software version, device status) on the vehicle can also be sent to the server. This allows the server to work with detailed status information of all devices of the whole fleet and even invoke device specific maintenance functionality based on their capabilities.

Important events of what happens on the vehicle (e.g. door release, emergency calls, stop-requests, automatic announcements, etc.) can be transmitted with the vehicleevent message.

To analyze issues on the on board systems there are messages to request screenshots of the contents of the displays or log-files of any device on the vehicle.

Maturity

The Diagnostics-Service maturity level is currently considered to be [Draft](#).

Used Messages

This service uses the following messages:

- Generic system status and client information transmission: **systemstatus ('systemstatus Element' in the on-line documentation)**, and **systemstatusreply ('systemstatusreply Element' in the on-line documentation)**. (These already exist in MPI 1.0 and are

- considered mature)
- Device status transmission: **devicestatusupdate** ('**devicestatusupdate Element**' in the **on-line documentation**) and **devicestatusupdatereply** ('**devicestatusupdatereply Element**' in the **on-line documentation**) and additional messages:
 - Screenshot requests and transmission: **requestsscreenshot** ('**requestsscreenshot Element**' in the **on-line documentation**) and **requestsscreenshotreply** ('**requestsscreenshotreply Element**' in the **on-line documentation**)
 - Logfile requests and transmission: **requestlogfiles** ('**requestlogfiles Element**' in the **on-line documentation**) and **requestlogfilesstatus** ('**requestlogfilesstatus Element**' in the **on-line documentation**).
 - Device restarts: **requestdevicerestart** ('**requestdevicerestart Element**' in the **on-line documentation**) and **requestdevicerestartconfirmation** ('**requestdevicerestartconfirmation Element**' in the **on-line documentation**)
 - Enabling a device test-mode: **requestdevicetestmode** ('**requestdevicetestmode Element**' in the **on-line documentation**) and **requestdevicetestmodeconfirmation** ('**requestdevicerestartconfirmation Element**' in the **on-line documentation**)
 - Important event transmission: **vehicleevents** ('**vehicleevents Element**' in the **on-line documentation**) and **vehicleeventsreply** ('**vehicleeventsreply Element**' in the **on-line documentation**).

Which features of the specification a given server or client implements is up to the implementer and the project requirements. As long as both parties know which of these messages are used in a given project there are no issues.

Communication

The communication flow is in both directions. For the status and informational messages the client is expected to send the information as soon as it happens or in regular intervals. For the cases where the train should send specialized information there is a request message that the server may issue to the client and the client should then answer with the requested information or confirm that the command was received.

Requirements

The client should make sure that the systemstatus is sent at least once per system-startup. This may either be once per boot or once per startup of the process that implements the mpi-interface. This allows the server to get to know about software version changes and other status changes.

Similarly the vehicle should send devicestatusupdate-messages as soon as the state of a device changes.

For the transmission of vehicle events the requirements are the same as for the transmission of

gpsdata: The events should always be sent in chronological order and if there are many events to send (because of connection issues) the buffered events should be sent in batches of 20-100 messages each.

3.4.11 Driver Message

The Driver message service allows to send messages to the driver. These messages should be displayed to the driver and may contain important information.

The server may request a confirmation for a message and the client should then let the user confirm the reception of the message via at least an additional button to confirm that he read the message.

Maturity

The DriverMessage-Service maturity level is currently considered to be [Mature](#).

Used Messages

This service uses the messages **drivermessage** ('**drivermessage Element**' in the **on-line documentation**) to send a text message to the driver and **drivermessagestatus** ('**drivermessagestatus Element**' in the **on-line documentation**) so that the client can update the server regarding the status of a received drivermessage.

Communication

The communication flow for the Driver message service is mainly from server to client. The server sends a drivermessage to the client if there is a message and the client responds with the drivermessagestatus message.

Requirements

3.4.12 Configuration

The configuration service provides generic methods for the server to send configuration updates to the train.

It uses an abstract concept of configuration targets which can be updated from the server. A configuration target may be any place the client may save configuration data: A configuration-file, a folder with multiple files, a database or anything else.

It is important to note that there are no globally defined targets and the available targets and their corresponding data format must be mutually agreed upon between the implementers of server and the train. So additional specification work is needed for this to work. Often such configuration data is already in use by software running on the train. MPI helps by defining a standard for how such configuration data can be transmitted and if it is implemented once on both sides it allows for easy extension with additional configuration possibilities.

It also allows configuration updates to take place at a certain point in the future so that bigger updates can already be transferred to the train but activated later on at a specific point in time. (For example, time table change).

Maturity

The Configuration-Service maturity level is currently considered to be [Draft](#).

Used Messages

This service uses the messages **configupdate** ('**configupdate Element**' in the **on-line documentation**) and **configupdatestatus** ('**configupdatestatus Element**' in the **on-line documentation**). The server sends a configupdate message to the client if it has new configuration data for a certain target. The client then replies with multiple configupdatestatus-messages to update the server on the process of applying that configupdate (similar to the schedule update mechanism).

Communication

The communication flow for the configuration service is mainly from client to server as the server initiates the configuration updates.

Requirements

The server should maintain its own information about which config targets a given client has in which version. This allows the server to detect any config changes that need to be transmitted to a client.

The server is free to implement mechanisms to decide which train should be updated with which configuration version at which time. Possible modes would be: The user decides which configuration to send to which train, the server having different channels (e.g. beta and stable) and a mapping between trains and their channels und use that to decide which version is active for which train, or simply send all configuration data updates to all trains automatically, etc.

3.4.13 Passenger Counting

The passenger counting service provides the ability to send passenger counting data and passenger occupancy from the client to the server.

It is up to the implementers to decide if only passenger counting or passenger occupancy data is transmitted in a project, or if both data is exchanged. As every message type serves its own purpose and can be used in different scenarios there is no requirement to implement both types of messages.

Maturity

The PassengerCounting-Service maturity level is currently considered to be [Draft](#).

Used Messages

This service uses the messages **passengercounts** ('**passengercounts Element**' in the **on-line documentation**), **passengercountsreply** ('**passengercountsreply Element**' in the **on-line documentation**) and **passengeroccupanciesupdate** ('**passengeroccupanciesupdate Element**' in the **on-line documentation**) and **passengeroccupanciesupdatereply** ('**passengeroccupancyupdatereply Element**' in the **on-line documentation**).

Communication

The communication flow for this service is from client to server. The vehicle sends passenger counting data to the server and the server acknowledges the received data.

Requirements

The passenger counting service currently has no special requirements that need to be fulfilled by implementators.

3.5 Transports

3.5.1 WebSocket Transport

- **Introduction**

- **Hostname and Port**
- **SSL-Support**
- **Connection management**
- **Messaging basics**

Introduction

The WebSocket transport is the main transport to transfer the MobilePI-Data between server and client.

It is based on the official [Websocket specification](#).

Hostname and Port

The server is normally reached via a hostname defined beforehand. The port is not a standard HTTP-Port but a different one. For clients implementing the protocol it must be possible to configure to which hostname and port the client should connect.

SSL-Support

The client must be able to connect to the DiLoc|Rail WebSocket server via SSL and with a standard unencrypted connection. For testing purposes, the server is normally run without SSL, and later when the production server is set up SSL-Support is enabled.

Connection management

The connection between the DiLoc|Rail server and the client is always established from the client. In mobile networks (GPRS/3G/4G) it is normally not possible to establish a connection to a mobile device from the internet because all mobile devices are in private networks behind gateways. In order to have a persistent connection to the server the client must always try to reconnect to the server as soon as it detects a connection failure. Normally it is difficult to detect network failures automatically when they happen. So if the client is not sending data in regular intervals it should at least send a ping-message regularly to be able to detect network problems reliably.

Messaging basics

For ease of use the protocol is basically stateless. That means there is no need to send any handshake messages after setting up a connection. If the connection is established the server and the client should be ready to send and receive any valid message.

Each WebSocket Message should contain exactly one **XML-message ('message Element' in the on-line documentation)**. The message-tag is the common wrapper around everything that is exchanged between server and client. Each message contains exactly one tag that identifies


the data that is sent in the message. For each tag that is sent in either direction there normally exists a corresponding reply-tag that is used to reply to a given message. For example, the **gpsdata** ('**gpsdata Element**' in the on-line documentation)-tag is sent from the client to the server and the server replies with a **gpsdatareply** ('**gpsdatareply Element**' in the on-line documentation)-tag. There are tags that may be sent in each direction and there are tags that are only sent in one specific direction. The schema documentation explains in which direction a certain tag may be sent.

Processing on the Server

1. When a message is received at the server it is first checked if the message contains valid XML. If not, a **reply** ('**reply Element**' in the on-line documentation) is sent with an **error** ('**error Element**' in the on-line documentation).
2. If the xml is valid it may be validated against the XML-Schema to prove that it is valid. If it is not valid the corresponding reply-tag is sent back containing the error of type validation.
3. If the xml could be validated, the auth-hash is checked against all valid auth hashes to be able to authenticate the message coming from a valid connection partner and device. If not a corresponding reply-tag is sent back containing an error of type authfail.
4. If the auth-hash could be validated the message is processed by the server. While processing the message it could be that some of the received data is invalid, in this case an error of type datainvalid is sent back to the client. If something other happens on the server an error of type fail is sent to the client.
5. If everything could be processed as expected a reply with a success tag is sent back.

Asynchronicity

Since all message-replies can be assigned to a corresponding message the protocol allows for multiple messages of **different** types to be sent directly after each other without waiting for a reply. On the other hand this means that the next message of the same type must only be sent after a reply to the last message was received. So when a gpsdata-message is sent, the client must wait for a gpsdatareply-message from the server before the next gpsdata-message can be sent. But it is for example possible to send a ping-message (or any other message) while waiting for the gpsdatareply.

 When sending multiple messages of different type shortly after each other it may be difficult to correlate the corresponding message if the XML is invalid and not parsable by the server, because in this case it would be impossible to send the correct reply-tag back to the client.

4 DiLoc JSON Schedule Format

4.1 Introduction

The DiLoc JSON Schedule Format is a [JSON](#)-based format that describes the schedule of trains like they are provided for the DiLoc|OnBoard software.

The format mainly consists of an array of trains and is documented in detail in [DiLoc JSON Schedule Format Schema](#).

Trains

Every train contains information about the train (like train-number, departure-time etc,) and all the stations the train will travel to.

The train also contains a script (Drehbuch in German) that controls the different passenger information channels (like indoor tft displays, side and front led displays, audio equipment, etc).

Script

The script contains information that controls what a train outputs in certain situations. It is a sub-object of the train object. A script is contained in every train schedule.

The script-object contains an array of tracks.

Track Objects

Every track object contains the information that should be output on the corresponding channel. Every track object contains the following members:

- **type**: The type identifying the track/channel.
- **identifier**: The identifier that identifies the channel. This is unique per script. It allows a channel to have identifier-specific configuration.
- **startCommands**: An array of commands that should be processed when the train is bound.
- **stations**: An array of station-related commands that should be processed

Station Objects

The station-object is repeated in the stations-array. All stations are ordered in the direction of travel. It begins with the starting station and ends with the destination station.

It contains the following members:

- station: The ID-Code of the station that this object belongs to.
- commands: An array of command-objects that should be processed by the channel

Command Objects

Command objects are the heart of the script. Commands specify what a given channel should output at any given time. It contains the following members:

- name: The name of the command.
- [params]: The parameters to the command, may be non-existing if the command needs no parameters.
- on: Where the command should be triggered. This is either a negative number or a positive number meaning before and after a station. There are also some constants possible that are explained in the schema.

The data contained in the params-object depends upon the channel-type and the name of the command.

Every channel-type defines its own set of commands and command-parameters it understands.

The Track-type "tft"

This explains all commands that are allowed in the track type "tft".

Command showEmpty

This command shows an empty pearl-chain. This means the pearl-chain view is used but with empty data. This has the effect that the blue bars are shown at the top and bottom but there is no text content shown.

Command showPearlChain

Shows the pearl-chain. This shows the current pearl-chain based on the current position. In contrast to other commands of other channels this is an intelligent command that auto-updates the pearl-chain in case something happens (like advancing to the next station, etc).

Command showConnections

This shows the connections at the station the command is contained in. This is also an intelligent command as it automatically uses the connections defined in the train schedule or updated schedules if they are available.

Command showMessage

This shows a special message. This will be used for predefined messages. How these will be handled exactly is TBD.

Command showBlack

This shows a black screen and can be used if no train is selected, etc.

The Track-type "led"

This explains all valid commands in the track type "led". This track-type is used to control LED-displays.

Command showText

Shows the given text on the LE-Displays. What happens with text that is too long must be configured on the client side. There are probably two ways to handle longer text: Either scroll the text, or cut it off. Fonts to use and Font-sizes are also to be configured on the client level and are on purpose not contained in the script because the server generating the script should not need to care about this.

The allowed parameters are documented in detail in the schema.

Command showDestination

Shows the given destination information on the LE-Displays.

How the passed information is displayed for a given track is client dependent. For example a front-led normally never shows via-stations and just ignores the information. The line is normally shown in inverted colors on the left side. The destination is normally shown centered. All this information is configured on the client level and is nothing that can be controlled from the script.

The allowed parameters are documented in detail in the schema.

Command clear

Clears the display and outputs nothing.

The track-type "audio"

This explains all valid commands in the track-type "audio". This track-type is used to control audio output in trains.

Command enqueueText

This command enqueues a text that should be spoken via TTS (Text To Speech).

The allowed parameters are documented in detail in the schema.

Command enqueueFiles

This command enqueues a list of files that should be played in order. This is used to play files of

pre-recorded announcements.

The allowed parameters are documented in detail in the schema.

Command `clearQueue`

This command clears the current queue. If there is currently an announcement ongoing it is not aborted.

Example Train

This example shows a train traveling from St. Gallen (SGAB) to Trogen (TROG) via Speicher (SPEI). In reality there are more stations in between these stations but for simplicity reasons they were omitted.

The script contains four tracks:

- One of type "led" for the side-led's.
- One of type "led" for the front/back-led's
- One of type "tft" for the indoor tft-displays.
- One of type "audio" for the announcements during the drive.

```
{
  "trains": [
    {
      "trainNumber": "4711.15a",
      "script": {
        "tracks": [
          {
            "type": "led",
            "identifier": "side",
            "startCommands": [
              {
                "name": "showDestination",
                "params": {
                  "line": "S21",
                  "destination": "Trogen",
                  "via": [
                    "Speicher"
                  ]
                }
              }
            ]
          }
        ],
        "stations": [
          {
            "station": "SPEI",
            "commands": [
              {
                "name": "showDestination",
```

```

        "params": {
            "line": "S21",
            "destination": "Trogen",
            "via": []
        },
        "on": "-200"
    }
]
},
{
    "station": "TROG",
    "commands": [
        {
            "name": "showDestination",
            "params": {
                "line": "S21",
                "destination": "St. Gallen",
                "via": [
                    "Speicher"
                ]
            },
            "on": "-200"
        }
    ]
}
]
},
{
    "type": "led",
    "identifier": "front",
    "startCommands": [
        {
            "name": "showDestination",
            "params": {
                "line": "S21",
                "destination": "Trogen",
                "via": [
                    "Speicher"
                ]
            }
        }
    ],
    "stations": [
        {
            "station": "TROG",
            "commands": [
                {
                    "name": "showDestination",
                    "params": {
                        "line": "S21",
                        "destination": "St. Gallen"
                    },
                    "on": "-200"
                }
            ]
        }
    ]
}
]
}

```

```
    }
  ]
}
],
{
  "type": "tft",
  "identifrier": "pis",
  "startCommands": [
    {
      "name": "showPearlChain"
    }
  ],
  "stations": [
    {
      "station": "SPEI",
      "commands": [
        {
          "name": "showConnections",
          "on": "-500"
        },
        {
          "name": "showPearlChain",
          "on": "halt"
        }
      ]
    }
  ]
},
{
  "type": "audio",
  "identifrier": "default",
  "stations": [
    {
      "station": "SGAB",
      "commands": [
        {
          "name": "enqueueText",
          "params": {
            "text": "Herzlich Willkommen in der Trogener Bahn nach Trogen.",
            "lang": "de"
          }
        },
        {
          "on": "100"
        }
      ]
    }
  ],
  {
    "station": "SPEI",
    "commands": [
      {
        "name": "enqueueText",
        "params": {
          "text": "Nächster Halt Speicher.",

```

```

        "lang": "de"
      },
      "on": "-400"
    }
  ]
},
{
  "station": "TRO",
  "commands": [
    {
      "name": "enqueueText",
      "params": {
        "text": "Nächster Halt Trogen. Dieser Zug endet hier. Wir bitten
alle Fahrgäste auszusteigen und bedanken uns für die Fahrt mit der Trogener Bahn.",
        "lang": "de"
      },
      "on": "-500"
    }
  ]
}
]
}
]
},
"departureTime": "15:34:30",
"nextTrainNumber": "4711",
"travelsOnDays": [
  "mo",
  "fr",
  "sa",
  "su"
],
"stations": [
  {
    "station": "SGAB",
    "timeTableDepartureTime": 0,
    "stopType": "departure"
  },{
    "station": "SPEI",
    "timeTableArrivalTime":7200,
    "timeTableDepartureTime":7290,
    "stopType": "halt",
    "connections": [
      {
        "departureTime": "2014-07-06 15:45:00",
        "destination": "Katzwiler",
        "category": "bus",
        "line": "130",
        "platform": "B"
      }
    ]
  },{
    "station": "TROG",

```

```
        "timeTableArrivalTime": 14000,  
        "stopType": "arrival"  
    }  
  ]  
}  
]
```

4.2 Schema

This contains the complete [JSON Schema](#) of the DiLoc JSON Schedule format.

Currently this is hosted online so you will only see this if you have an internet connection.

5 Index

Audio Control, 21-22

Authentication, 11-12

Booklet Cover Page, 0

Change Log, 6-10

Diagnostics, 22-24

Display Control, 21

Driver Message, 24

General Information, 3

GPS Service, 12-13

Introduction, 29-36

Introduction to the Mobile-PI-Interface, 4-6

Logical Positioning, 13-15

MobilePI

Authentication, 11-12

Change Log, 6-10

General Information, 3

Introduction to the Mobile-PI-Interface, 4-6

Service Maturity, 10-11

On demand schedules, 20-21

Schema, 36

Service Maturity, 10-11

SSL, 26-28

Stop On Demand, 18-19

Train Binding, 15-17

WebSocket, 26-28

WebSocket Transport, 26-28