TECHNISCHE UNIVERSITÄT WIEN

Institut für Fertigungstechnik und Photonische Technologien

Diplomarbeit

Konzepte zur Integration von FTS in die Materialflüsse der Produktion

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs (Dipl.-Ing. oder DI) unter der Leitung von

Univ.-Prof. Dr. Burkhard Kittl

(Institut für Fertigungstechnik und Photonische Technologien)

Dipl.-Ing Thomas Trautner

(Institut für Fertigungstechnik und Photonische Technologien)

eingereicht an der Technischen Universität Wien

Fakultät für Maschinenwesen und Betriebswissenschaften

von

Yilmaz Güzel, B.Sc.

00928979 (066 482)

Wien, im Oktober 2020

_____

Yilmaz, Güzel B.Sc.

Ich nehme zur Kenntnis, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung

Diplomarbeit

nur mit Bewilligung der Prüfungskommission berechtigt bin.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass die vorliegende Arbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen von mir selbstständig erstellt wurde. Alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, sind in dieser Arbeit genannt und aufgelistet. Die aus den Quellen wörtlich entnommenen Stellen, sind als solche kenntlich gemacht.

Das Thema dieser Arbeit wurde von mir bisher weder im In- noch Ausland einer Beurteilerin/einem Beurteiler zur Begutachtung in irgendeiner Form als Prüfungsarbeit vorgelegt. Diese Arbeit stimmt mit der von den Begutachterinnen/Begutachtern beurteilten Arbeit überein.

Wien, im Oktober 2020

_____

Yilmaz, Güzel B.Sc.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Diplomarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Univ.Ass. Dipl.-Ing. Thomas Trautner, der meine Diplomarbeit betreut und begutachtet hat und stets bei Fragestellungen, mich mit dem richtigen Rat, zuverlässig und hilfreich unterstützt hat. Bedanken möchte ich mich ebenfalls bei Univ. Prof. Dipl.-Ing. Dr. techn. Burkhard Kittl, der diese Arbeit am Institut für Fertigungstechnik an der Technischen Universität Wien ermöglicht hat.

Weiters danke ich Dipl.-Inform. Stefan Walter vom Frauenhofer IML, für die Beantwortung fachlicher Fragen im Bereich der Software openTCS. Abschließend möchte ich mich bei meinen Freunden, meiner Freundin und Familienmitgliedern bedanken, welche mir während des Studiums zur Seite standen.

Kurzfassung

Für die diskrete Fertigung, die durch eine Produktionsumgebung charakterisiert wird, in der Produkte als abzählbare Einheiten hergestellt werden, wird die Forderung nach kurzen Durchlaufzeiten, geringen Beständen und hoher Flexibilität zunehmend größer. Dies führt dazu, dass der innerbetriebliche Materialfluss als integratives Element im Unternehmen immer mehr an Bedeutung gewinnt. Fahrerlose Transportsysteme (FTS) bieten ein hohes Maß an Flexibilität, automatisieren den innerbetrieblichen Materialfluss und erfüllen vermehrt Aufgaben mit erhöhter Komplexität. Somit können Herausforderungen bzgl. variablen Produkten und kleinen Stückzahlen in einer sich ständig ändernden Produktionsumgebung bewältigt werden.

Das Ziel dieser Arbeit ist es, Konzepte für die Integration von FTS in die Prozesse der Materialtransporte zwischen Lägern und Produktion bzw. innerhalb der Produktion zu beschreiben. Zusätzlich soll in einer bestehenden MES Infrastruktur eine prototypische Implementierung zur Validierung der theoretischen Annahmen dienen.

Dazu steht im Zuge eines Forschungsprojektes der TU Wien in der Pilotfabrik ein mobiler Roboter samt Steuerungssoftware für den Materialtransport zur Verfügung. Als übergeordneter Flottenmanager wird die Open-Source Software openTCS verwendet, welche im Stande ist, neue Fahrzeuge einzubinden und einen Parallelbetrieb von Fahrzeugen unterschiedlicher Hersteller in demselben Arbeitsumfeld erlaubt.

Um die Schnittstelle des Flottenmanagers zu übergeordneten Systemen, wie zu einem MES, darstellen zu können, wurde die Software Node-Red verwendet. Im Praxisteil wird anschließend vorgeführt, wie über Node-Red ein Austausch von Datentelegrammen, etwa z. B. Transportaufträge oder Statusmeldungen, mit dem Flottenmanager openTCS stattfinden kann.

Ein weiterer Aspekt, welcher in dieser Arbeit behandelt wird, ist die Schnittstelle zur Kommunikation zwischen fahrerlosen Transportfahrzeugen (FTF) und einer FTS-Leitsteuerung. Dazu wird die VDA 5050 vorgestellt, in der Möglichkeiten für die Definition einheitlicher Schnittstellen zwischen der Leitsteuerung und den FTF aufgezeigt werden. Die Umsetzung erfolgt im Praxisteil, in dem ein Fahrzeugtreiber in openTCS programmiert wird, welcher die Daten des Fahrzeuges einliest und die entsprechenden Fahrbefehle nach Vorgabe der VDA 5050 versendet. Die Schnittstelle zum Fahrzeug wurde dabei durch MQTT realisiert.

Das Ergebnis dieser Arbeit soll zeigen, wie ein FTS in die IT-Architektur eines Produktionsunternehmen integriert werden kann, mit dem Fokus auf die Schnittstelle zum MES, und weiters beschreiben wie die Einbindung von zukünftigen Fahrzeugen erfolgen kann.

Abstract

For discrete manufacturing, which is characterized by a production environment in which products are manufactured as countable units, the demand for short turnaround times, low stocks and high flexibility is growing. As a result, the internal flow of materials is becoming increasingly important as an integrative element in the company. Automated Guided Vehicle (AGV) systems offer a high degree of flexibility, automate in-house material flow and increasingly perform tasks with increased complexity. This enables challenges regarding variable products and small quantities to be overcome in an ever-changing production environment.

The aim of this work is to describe concepts for the integration of AGV System into the processes of material transport between warehouse and production or within production. In addition, a prototypical implementation of the validation of theoretical assumptions is to be used in an existing MES infrastructure.

During the research project at TU Wien, a mobile robot with control software for material transport is available in the pilot factory. The open source software openTCS is used as the higher-level fleet manager, which can integrate new vehicles and allow parallel operation of vehicles delivered by different manufacturers in the same working environment. The software Node-Red was used to represent the interface of the fleet manager to higher-level systems, such as an MES. In the practical part, it is then demonstrated how an exchange of data telegrams, e. g. transport orders or status messages, can take place via Node-Red with the fleet manager openTCS.

Another aspect covered in this work is the interface for communication between an AGV and a Fleet Manager. For this purpose, the VDA 5050 is presented, which shows possibilities for the definition of uniform interfaces between the control system and the AGV. The implementation takes place in the practical part, in which a vehicle driver is programmed in openTCS, which reads the data of the vehicle and sends the corresponding driving commands according to the specification of the VDA 5050. The interface to the vehicles was realized by MQTT.

The result of this work is intended to show how an AGV can be integrated into the IT architecture of a production company, with a focus on the interface to the MES, and further describe how the integration of future vehicles takes place.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AGV	Automated Guided Vehicle
API	Application Programming Interface
BDE	Betriebsdatenerfassung
Bsp.	Beispiel
bspw.	beispielsweise
bzgl.	bezüglich
bzw.	beziehungsweise
ca.	circa
D2D	Device to Device
d.h.	das heißt
ERP	Enterprise Ressource Planning
etc.	et cetera
evtl.	eventuell
FTF	fahrerlose Transportfahrzeuge
FTS	fahrerlose Transportsysteme
ggf.	gegebenfalls
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
KCC	Kernel Control Center
KI	Künstliche Intelligenz
LAM	Lastaufnahmemittel
M2M	Machine – to - Machine
MDE	Maschinendatenerfassung
MES	Manufacturing Execution System
MQTT	Message Queuing Telemetry Transport
o. g.	oben genannt
openTCS	open transportation control system
QoS	Quality of Service
QR	Quick Response
REST	Representetional State Transfer
RFID	Radio-frequency identification
SLAM	Simultaneous Localization and Mapping
sog.	sogenannt
SPS	Speicherprogrammierbare Steuerung

SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TS	Transportstück
u. a.	unter anderem
URL	Uniform Resource Locator
vgl.	vergleiche
XML	Extensible Markup Language
z. B.	zum Beispiel

1 Einleitung

1.1 Problemstellung

In der diskreten Fertigung, in der Produkte als abzählbare Einheiten hergestellt werden, kann sich die Produktion von einer variantenreichen Serienfertigung bis hin zur Fertigung in kleinsten Stückzahlen bzw. bis zur Losgröße eins ändern. Kriterien wie Flexibilität, Robustheit sowie Erweiterbarkeit bzw. Veränderbarkeit werden daher für den innerbetrieblichen Materialfluss immer bedeutender. Um diesen steigenden Anforderungen gerecht zu werden, wird in der Intralogistik verstärkt auf die Vollautomatisierung gesetzt. Ein Beispiel für die Automatisierung des innerbetrieblichen Materialflusses sind FTS.

FTS werden bereits seit Anfang der 60er Jahre erfolgreich zur Automatisierung des innerbetrieblichen Materialflusses eingesetzt. Dabei wurden die möglichen Einsatzgebiete und Funktionalitäten fortlaufend weiterentwickelt. Infolgedessen wurde in der Industrie eine Vielzahl von Konzepten zur Integration von FTS in die Prozesse der Materialflüsse erstellt [1].

Die Einführung eines FTS für den innerbetrieblichen Materialfluss kann für ein produzierendes Unternehmen jedoch eine Herausforderung darstellen. Dabei spielt die Implementierung des FTS in die IT-Architektur des Unternehmens eine entscheidende Rolle. Insbesondere die Schnittstelle zum Manufacturing Execution System (MES), welche meist als übergeordnetes Host-System dient, kann sich als problematisch darstellen. An den Schnittstellen können Konflikte und Kommunikationsprobleme entstehen, sodass ein reibungsloser Austausch von Daten nicht gewährleistet wird. Folglich kann ein mangelhafter Informationsfluss die Prozesse in der Produktion beinträchtigen.

Ein weiteres Problem ist, dass viele FTS-Hersteller dem Anwender ihre Produkte und Know-how über proprietäre Lösungen anbieten. Dies führt einerseits zu einer Abhängigkeit und damit verbundenen hohen Folgekosten, anderseits können Produkte von anderen FTS-Herstellern schwer in eine bereits vorhandene FTS-Leitsteuerung integriert werden. Jedoch kann ein Anbieter bei größeren Systemen, wie in der Automobilindustrie nicht alle Anforderungen erfüllen, weshalb eine standardisierte Schnittstelle auch immer mehr an Bedeutung gewinnt. Somit ist die Einbindung von herstellerunabhängigen Fahrzeugen und die Einführung einer standardisierten Schnittstelle, ein weiteres Problem bei der Integration eines FTS.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, einen Einblick zu gewinnen, wie ein FTS in ein Produktionsunternehmen integriert werden kann. Hierbei wird gezeigt wie die Schnittstelle für den Austausch von Daten zwischen einem MES und einem FTS, ausgeführt werden kann und die dazu entsprechenden Dateninhalte und -formate beschrieben. Zusätzlich soll diese Arbeit Aufschluss darüber geben wie eine standardisierte Schnittstelle zu den Fahrzeugen realisiert werden kann, um die Einbindung von zukünftigen und herstellerunabhängigen Fahrzeugen zu ermöglichen.

Für die Validierung der theoretischen Annahmen, wird im Zuge der Pilotfabrik der TU Wien eine prototypische Implementierung anhand eines Testfeldes durchgeführt. Innerhalb dieses Testfeldes der Pilotfabrik wird ein mobiler Roboter mit entsprechender Steuerungssoftware für den Materialtransport eingesetzt (siehe Abbildung 1.1).

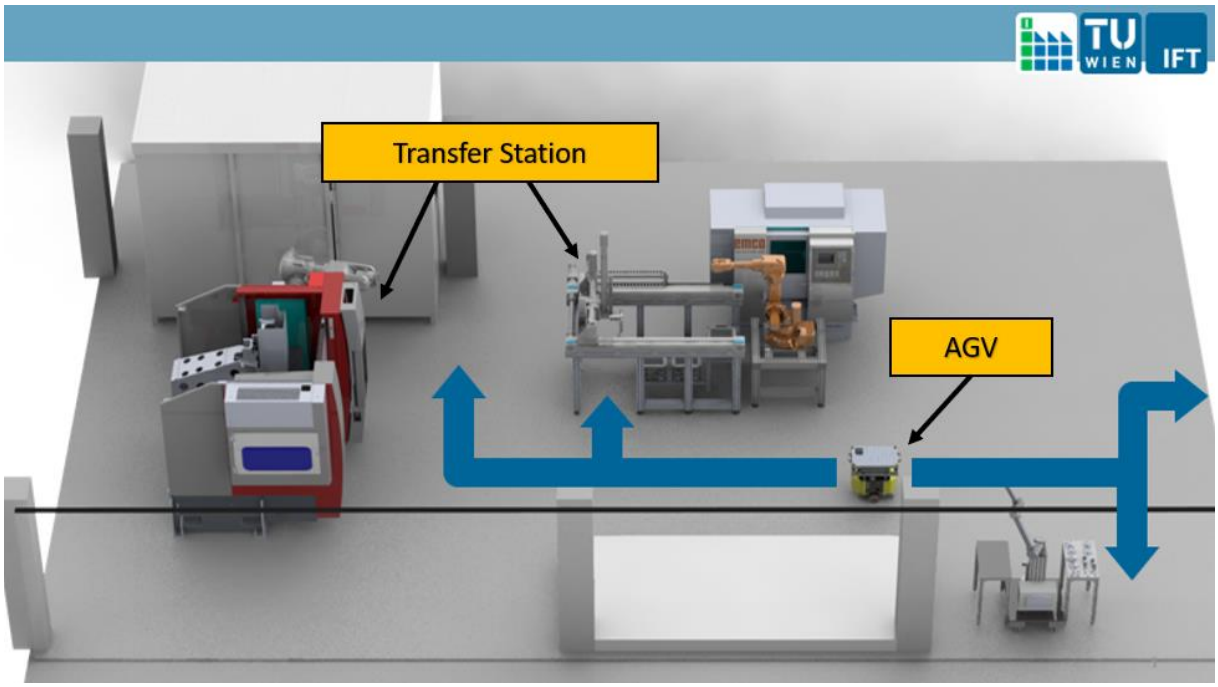

Abbildung 1.1: Anlagenlayout des Testfeldes der TU Wien Pilotfabrik, von Thomas Trautner (Abdruck mit Genehmigung).

Da die Steuerungssoftware ausschließlich zum Verwalten eines Fahrzeuges konzipiert ist, soll nach Möglichkeit ein übergeordneter Flottenmanager implementiert werden, welcher im Stande ist, neue Fahrzeuge einzubinden und diese im weiteren Verlauf zu verwalten und zu steuern. Über den Flottenmanager werden weiters Versuche hinsichtlich einer Kommunikation mit einem MES betrieben, um so den Austausch von Datentelegrammen wiedergeben zu können.

1.3 Vorgehensweise

Zum besseren Verständnis wird zu Beginn eine theoretische Recherche durchgeführt und die Ergebnisse in Form einer Grundlagentheorie wiedergegeben. Dabei werden wichtige Aspekte eines FTS, wie die Fahrzeugsteuerung, Leitsteuerung, Navigation, Infrastruktur und periphere Systeme, behandelt. Anschließend werden die Themen Produktionsleitsystem und Produktionsplanung kurz behandelt. Das Augenmerk liegt dabei auf dem MES, welche in einer Produktionsumgebung überwiegend als Host-System für das FTS dient. Weiters wird im Theorieteil das Hypertext Transfer Protocol (HTTP), Message Queuing Telemetry Transport (MQTT) und JavaScript Object Notation (JSON) erläutert, welche im Rahmen des Praxisteiles zum Einsatz kommen.

Im nächsten Abschnitt werden Möglichkeiten vorgestellt, wie eine Schnittstelle zu übergeordneten Systemen umgesetzt werden. Des Weiteren wird beschrieben wie nach der VDA 5050 (VDMA) eine offene Schnittstelle zur Kommunikation mit den Fahrzeugen realisiert wird. Darauffolgend werden mögliche Softwarelösungen vorgestellt, welche derzeit am Markt verfügbar sind und für die Pilotfabrik in Betracht kommen.

Im Praxisteil wird ein Flottenmanager innerhalb des Testfeldes der Pilotfabrik implementiert. Hierzu wird auf Basis des vorherigen Kapitels ein Flottenmanager ausgewählt und eingesetzt, welcher die notwendigen Anforderungen erfüllt. Dabei wird die Aufgabe in zwei Teile gegliedert. Im ersten Teil soll demonstriert werden, wie eine Schnittstelle zur Kommunikation zwischen einem MES und dem Flottenmanager umgesetzt werden kann. Im nächsten Teil soll eine Schnittstelle zwischen dem Flottenmanager und einen mobilen Roboter programmiert werden. Die Kommunikation soll dabei über MQTT erfolgen.

Um die Schnittstelle zwischen einem MES und der FTS-Leitsteuerung zu beschreiben, wird die Software Node-Red verwendet, welche durch Austausch von Datentelegrammen, wie Statusabfragen oder Transportaufträge über unterschiedliche Protokolle, die Kommunikation mit der FTS-Leitsteuerung wiedergibt.

Als letzter Schritt wird eine Schnittstelle zum Fahrzeug realisiert. Dabei soll die Schnittstelle des Flottenmanagers so programmiert werden, dass der Datenaustausch mit dem Fahrzeug über einen MQTT Broker erfolgt. Die eingehenden Fahrzeugdaten sind bereits vorhanden, somit werden ausschließlich die Fahrbefehle nach Vorgabe der VDA 5050 umgesetzt.

2 Grundlagen

In diesem Abschnitt werden, zum besseren Verständnis des Praxisteils, theoretische Grundlagen zu den Themen behandelt, die während dieser Arbeit zur Verwendung gekommen sind.

2.1 Grundlagen fahrerlose Transportsysteme

FTS gelten in der Automatisierung des innerbetrieblichen Materialflusses als die Technologie, die dem Anwender ein Höchstmaß an Flexibilität bietet [2]. Die VDI 2510 definiert ein FTS wie folgt:

„*Fahrerlose Transportsysteme (FTS) sind innerbetriebliche, flurgebundene Fördersysteme mit automatisch gesteuerten Fahrzeugen, deren primäre Aufgabe der Materialtransport, nicht aber der Personentransport ist. Sie werden innerhalb und außerhalb von Gebäuden eingesetzt* “ [2].

Aufgrund der unterschiedlichen Anforderungen und Einsatzgebiete haben sich im Laufe der Zeit viele Konzepte betreffend FTS entwickelt. Die möglichen Systeme lassen sich größtenteils, wie in Tabelle 1 dargestellt, in folgende Eigenschaften charakterisieren [2]:

Tabelle 1: Typische Kennwerte eines FTS [2]

Anzahl FTF je System	ein bis mehrere hundert
Tragfähigkeit eines FTF	wenige kg bis über 50 t
Fahrgeschwindigkeit	typischerweise 1 m/s, abweichende Werte möglich
Fahrkurslänge	wenige m bis über 10 km
Anzahl der Stationen	unbegrenzte Anzahl an Lastwechsel- und Arbeitsstationen
Anlagensteuerung	manuell bis vollautomatisch oder in komplexe Materialflusssysteme integriert
Einsatzdauer	sporadisch bis „rund um die Uhr“
Antriebskonzepte	elektromotorisch, mit oder ohne Batterie, verbrennungsmotorisch

Im wesentlich besteht ein FTS, wie in Abbildung 2.1 dargestellt, aus folgenden Komponenten [3], [4]:

- einem oder mehreren FTF: sind an die jeweiligen Aufgaben individuell angepasst
- zentrale Leitsteuerung: übernimmt die Schlüsselrolle bei der Steuerung der einzelnen Komponenten und kann mit übergeordneten Systemen gekoppelt werden
- Kommunikationseinrichtung zur Datenübertragung: stellt die Verbindung und die Kommunikation mit der Leitsteuerung, den FTF und anderen Systembestandteilen sicher
- Navigationssystem: zur Standortbestimmung und Lageerfassung
- Infrastruktur und periphere Einrichtungen: beinhaltet u. a. stationäre Einrichtung zur Lageerfassung, Lasthandling und elektrische Energieversorgung

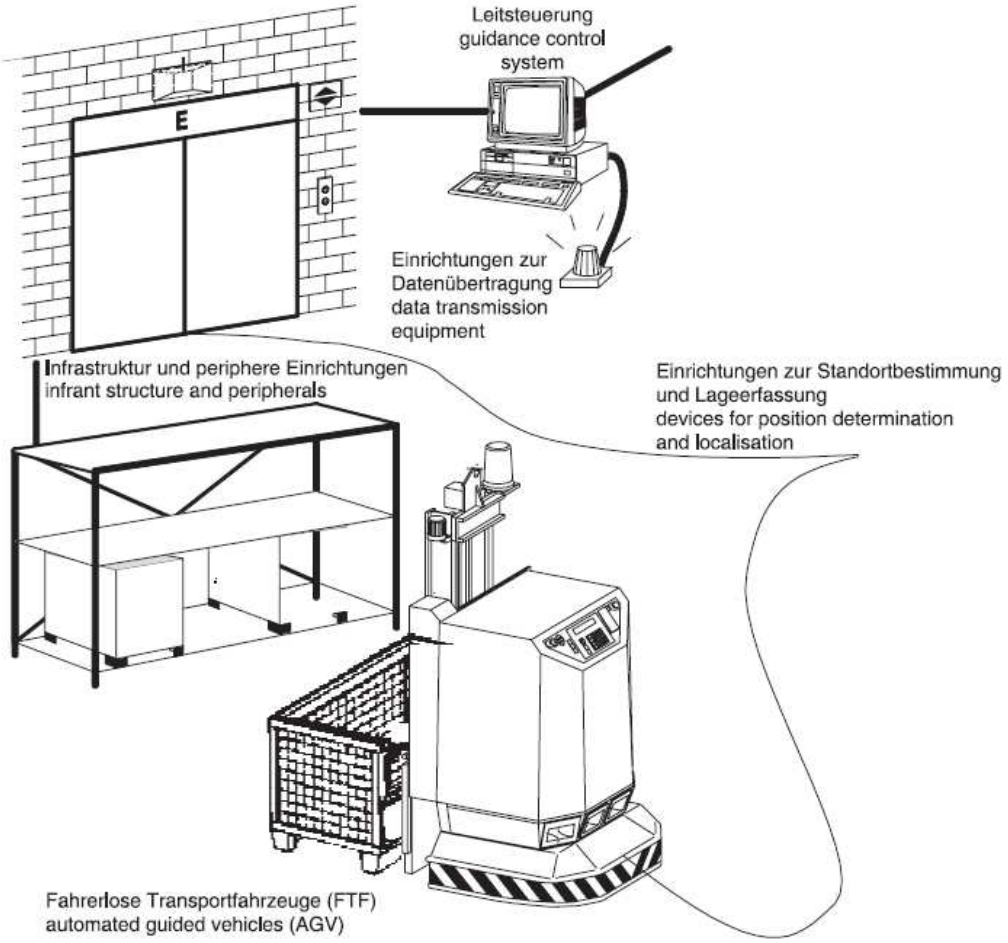

Abbildung 2.1: Komponenten eines FTS [2]

2.1.1 FTS-Leitsteuerung

Anspruchsvolle automatisierte Systeme, benötigen durchdachte Steuerungskomponenten. Im Falle eines FTS erfolgt diese in zwei Ebenen [5]:

- in der operativen Ebene (Fahrzeugebene) arbeitet die Fahrzeugsteuerung (siehe Kapitel 2.1.2)
- übergeordnet befindet sich die administrative Ebene (Leitsteuerung), welche das Gesamtsystem verwaltet und überwacht

Die Leitsteuerung ist das „Gehirn“ eines FTS. Sie übernimmt eine wichtige Schlüsselrolle des FTS und koordiniert den Materialfluss [4]. Nach der VDI 4451 wird eine FTS-Leitsteuerung wie folgt definiert:

„Eine FTS-Leitsteuerung besteht aus Hard- und Software. Kern ist ein Computerprogramm, das auf einem oder mehreren Rechnern abläuft. Sie dient der Koordination mehrerer Fahrerloser Transportfahrzeuge und/oder übernimmt die Integration des FTS in die innerbetrieblichen Abläufe“ [5].

Zu den Aufgaben einer FTS-Leitsteuerung gehören laut Dickmann [4]:

- das Integrieren des FTS in seine Umgebung
- das Annehmen von Transportaufträgen
- die Möglichkeit, dem Benutzer einen Service anzubieten
- Funktionsblöcke zu den entsprechenden Aufgaben bereitzustellen

Funktionsbausteine einer FTS-Leitsteuerung

Die Funktionen der Leitsteuerung können, wie in Abbildung 2.2 dargestellt, in folgende Gruppen eingeteilt werden [5]:

- Benutzer-Interface
- FTS-Interne Materialflusssteuerung
- Transportauftragsabwicklung
- Service-Funktionen

Bei einem hohen Komplexitätsgrad des Gesamtsystems können bestimmte Funktionen wegfallen bzw. von anderen Leitsystemen übernommen werden. Für die FTS-Leitsteuerung sind Transportauftragsabwicklung und das Benutzer-Interface ausschlaggebend. Das Benutzer-Interface ist die Schnittstelle zwischen dem Benutzer bzw. Auftraggeber des FTS und der Transportabwicklung sowie der FTS-internen Materialflusssteuerung bzw. den Service-Funktionen. Hierzu zählen Maschine-Maschine-Schnittstellen, wie etwa LAN- und WLAN-Protokolle, sowie auch Mensch-Maschine-Schnittstellen [5].

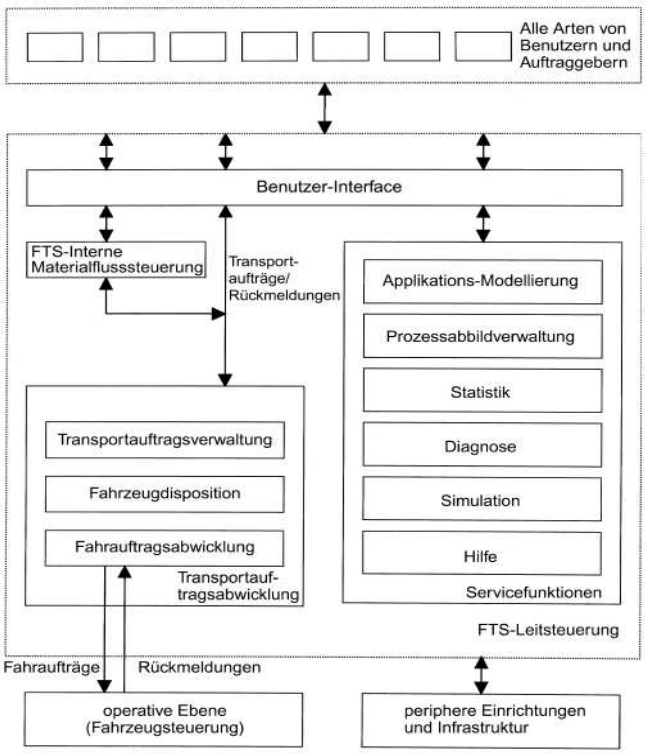

Abbildung 2.2: Funktionsbausteine einer FTS-Leitsteuerung [5]

Die Transportauftragsabwicklung sorgt dafür, dass das richtige Material zur richtigen Zeit am richtigen Ort bereitsteht. Die Basis dafür bilden die Transportaufträge. Transportaufträge bekommen eine eindeutige Kennzeichnung (ID). Neben dieser sind weitere Daten wie die Abholstelle (Quelle), das Auftragsziel (Senke) und evtl. eine Auftragspriorität mit enthalten. Die Auftragspriorität regelt den erlaubten Start- oder Endzeitpunkt eines Auftrags. Das benötigte Hintergrundwissen befindet sich in der internen Materialflusssteuerung. Mittels dieser Informationen werden Transportaufträge generiert. Zu den Funktionsbausteinen der Transportauftragsabwicklung gehören nach der VDI 4451 Blatt 7 [5]:

- die Transportauftragsverwaltung
- die Fahrzeugdisposition
- die Fahrauftragsabwicklung

In der Transportauftragsverwaltung sind die Transportaufträge meist in Reihenfolge ihrer Erzeugung beinhaltet. Bei Bedarf können die Transportaufträge auch mit einer zeitlichen Einplanung aufgelistet werden. Je nach Priorität der Aufträge werden diese von der Transportauftragsverwaltung kategorisiert und nach ihrer Durchführbarkeit getestet. In der Regel ist dies gegeben, wenn an der Quelle das Material zur Verfügung steht und an der Senke ein Leerplatz verfügbar ist. Die Durchführung kann ebenfalls

unterschiedlichen Kriterien unterliegen, wie etwa der Fahrzeugdisposition. In der Fahrzeugdisposition wird für den jeweiligen Transportauftrag das „günstigste Fahrzeug gewählt“, mehr dazu folgt in Kapitel 3.1.2 [5].

Von der Fahrzeugdisposition werden Transportaufträge an die Fahrauftragsabwicklung übermittelt. Der erteilte Fahrauftrag kann entweder komplett oder in Teilaufträgen an die Fahrzeugsteuerung übermittelt werden. Während der Auftragsausführung meldet das FTF kontinuierlich den Statuszustand an die Fahrauftragswicklung zurück. Somit werden Informationen aktualisiert und ggf. neue Teilaufträge erteilt. Bei Beendigung des Auftrags meldet die Fahrauftragsabwicklung den abgeschlossenen Auftrag an die Fahrzeugdisposition weiter. Zusätzlich übernimmt die Fahrauftragsabwicklung bei Anlagen mit mehreren Fahrzeugen auch die Verkehrsleitsteuerung. Ziel ist es, an Kreuzungen oder Knotenpunkten für einen geregelten Durchlauf zu sorgen und Blockaden oder Kollisionen zu vermeiden [5].

FTS-Leitsteuerungen der neuen Generation verfügen über eine umfassende Service-Funktion. Die Funktionsgruppe „*Service*“ umfasst kurzgefasst nach Albrecht [1] folgende Funktionen:

- Applikationsmodellierung: Modellierung des Layouts und/oder Aktivierung/Deaktivierung von einzelnen FTF
- Prozessabbildverwaltung: Anlagenvisualisierung, Mitteilung der Informationen wie Fahraufträge, Fahrzeugzustände und Position an den Benutzer
- Statistikfunktion: Sammeln und Auswerten von Daten über die Auslastung bzw. auftretende Fehler der Anlage
- Diagnosesystem: zur effizienten Fehlersuche und Erkennung
- Hilfsfunktionen: bieten eine unterstützende Funktion für den Benutzer
- Simulation: hier wird u. a. das Systemverhalten analysiert

Bediener und Auftraggeber

Ein Transportauftrag kann durch unterschiedliche Wege an eine Leitsteuerung übertragen werden, dazu zählen [5]:

- Bedienpersonal der Anlage bzw. Service- und Instandhaltungspersonal, z. B. via Terminal oder Monitor
- Anlagen-Bediener über ein Betriebsdatenerfassungssystem, hierzu zählen u. a. Ruftaster oder Barcode-Scanner
- Host-Rechnersysteme wie Enterprise Resource Planning (ERP), MES, Lagerverwaltungssysteme sowie Materialflussrechner
- Platz- und Belegmelder, Übergabestationen sowie weitere periphere Einrichtungen

Systemintegration einer FTS-Leitsteuerung

Je nach Komplexitätsgrad, Anzahl der Fahrzeuge und Anlagengröße kann die Anbindung des FTS unterschiedlich ausgeführt werden. Anhand des Beispiels aus Abbildung 2.3, wird die Anbindung einer High-End-Ausbaustufe eines FTS erläutert. Ein Multiserver-System und separate Bedienungs- und Visualisierungsrechner sind hier ein üblicher Standard. Neben einer sicheren Datenerhaltung ist ebenso eine Fernkommunikation über Modem oder Internet gegeben. Über drahtlose Kommunikationssysteme wird die Verbindung zu den Fahrzeugen hergestellt. Obligatorisch ist die Anbindung an periphere Sub-Systeme. Die Datenübertragung zu übergeordneten Host-Rechnern erfolgt häufig über lokale Ethernet-basierte Netzwerke mithilfe von Transmission Control Protocol/Internet Protocol(TCP/IP) oder über drahtlose Kommunikation. Über diese Kopplung zwischen dem Host und der FTS-Leitsteuerung können folgende Telegrammarten ausgetauscht werden [5]:

- Abgleichs-Telegramm
- Transportaufträge
- Transportquittungen
- Statustelegramme

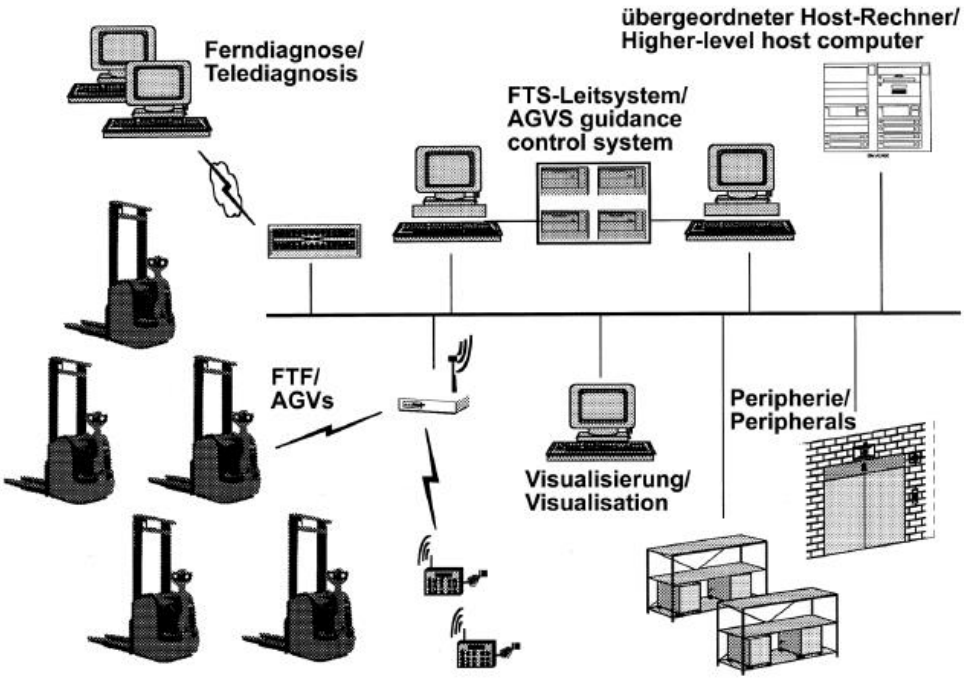

Abbildung 2.3: Struktur einer FTS-Leitsteuerung mit hohen Komplexitätsgrad [5]

2.1.2 Fahrerlose Transportfahrzeuge

Durch das große Anwendungsspektrum für FTS existiert eine Vielzahl an unterschiedlichen FTF-Bauarten auf dem Markt. Daher bieten die meisten Hersteller Grundsysteme an, welche nach Kundenwunsch modifiziert werden können [6]. Nach der VDI 2510 sind FTF wie folgt definiert:

„FTF sind flurgebundene Fördermittel mit eigenem Fahrantrieb, die automatisch gesteuert und berührungslos geführt werden. Sie dienen dem Materialtransport, und zwar zum Ziehen und/oder Tragen von Fördergut mit aktiven oder passiven Lastaufnahmemitteln“ [2].

Die Einteilung der FTF-Bauformen erfolgt abhängig von den transportierten Lasten. Ein weiteres Kriterium für die Kategorisierung von FTF bildet ihre Kapazität. Diese legt die Anzahl der Transportstücke fest, die gleichzeitig befördert werden können. Ein Single-Load-Carrier kann demnach nur ein Transportstück (TS) befördern. Hingegen ist es möglich mittels Multiple-Load-Carrier mehrere TS gleichzeitig zu transportieren [2]. Zusätzliches Merkmal eines FTF ist die Art der Lastaufnahme. Dabei wird prinzipiell zwischen Last ziehenden und Last tragenden FTF unterschieden. In Abbildung 2.4 sind die gängigsten FTF-Bauformen dargestellt. Neben diesen Bauformen existieren noch weitere FTF-Modelle, für diese wird auf die VDI 2510 [2] verwiesen.

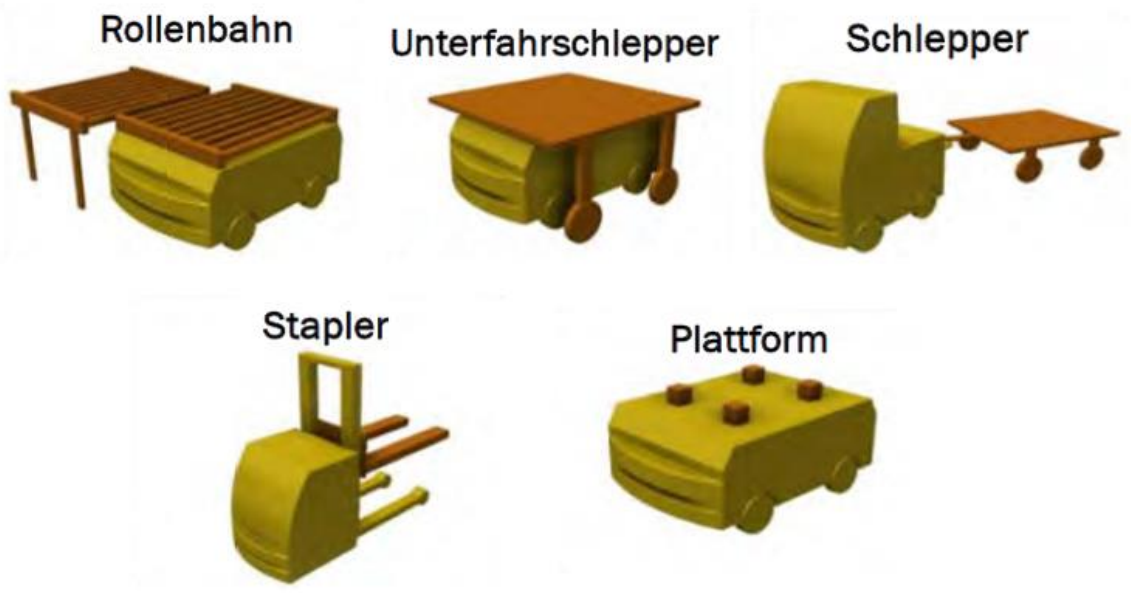

Abbildung 2.4: gängige FTF-Modelle [7]

Komponenten eines FTF

Die wesentlichen Komponenten eines FTF sind [2]:

- Fahrzeugsteuerung, Sensoren und Aktoren: sorgen für das Zurechtfinden des Fahrzeugs in der Umgebung.
- Energieversorgung: in Abhängigkeit vom Antriebskonzept kommen unterschiedliche Energieversorgungselemente zum Einsatz. Als Energiewandler werden vorwiegend Batterien verwendet.
- Warn- und Sicherheitseinrichtung: je nach Anwendung werden Blinkleuchten, Fahrtrichtungsanzeiger, akustische Warneinrichtung, etc. eingesetzt. Um Kollisionen zu vermeiden, werden heutzutage berührungslose Systeme wie Ultraschallsensoren oder Infrarot-/Laser-Scanner verwendet.
- Datenübertragung: in der Regel bestehen Sie aus einer fahrzeugseitigen mobilen und einer stationären Einrichtung.
- Bedienelemente: ermöglichen eine manuelle Bedienung des Fahrzeuges.
- Fahrwerk: bestimmt das Bewegungsverhalten der Fahrzeuge.
- Lastaufnahmemittel (LAM): kann aus einer Vielzahl von Standard-Lastaufnahmemitteln gewählt werden.

Fahrzeugsteuerung

Neben der FTS-Leitsteuerung ist die Fahrzeugsteuerung maßgeblich für die Flexibilität, Verfügbarkeit und Leistungsfähigkeit des gesamten FTS. Die Software der Fahrzeugsteuerung sollte gemäß der VDI-Richtlinie modular aufgebaut sein. Zu den Funktionen einer Fahrzeugsteuerung gehören primär die Standort- und Wegbestimmung des Fahrzeuges, Abwicklung von Transportaufträgen, Energiehaushalt und die Fahrzeugführung. Des Weiteren sind Funktionen wie Verkehrsregelung und Lasthandhabung in das System implementiert. Diese Funktionen müssen klar ersichtlich, autonom programmiert und austauschbar sein. Weiters sind Module für Notaus, Diagnose, Halbautomatik und manuellen Betrieb verfügbar. Je nach Anforderungen an die Fahrzeugsteuerung und des spezifischen Anwendungsfalles kommen unterschiedliche Hardware-Plattformen zum Einsatz, wie [8]:

- Einplatinenrechner
- Speicherprogrammierbare Steuerung (SPS)
- Mehrplatinenrechner
    - Bussysteme (seriell, parallel)
    - Rechnernetze

Sieht man eine Fahrzeugsteuerung als „Black Box“, so muss diese über geeignete Schnittstellen mit ihrer Umgebung kommunizieren können. Der Aufbau einer Fahrzeugsteuerung nach VDI 4451 Blatt 4 ist in Abbildung 2.5 dargestellt.

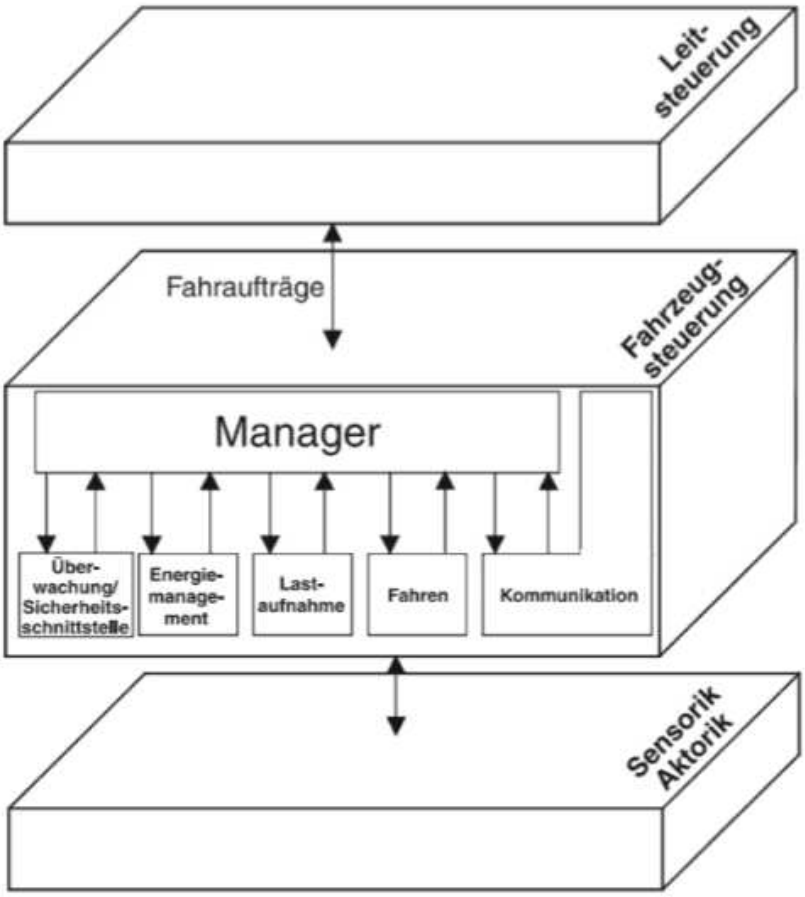

Abbildung 2.5: modular aufgebaute Fahrzeugsteuerung [8]

Zum internen Umfeld der FTF-Steuerung zählen LAM, Sensoren und Aktoren, das Bedienfeld am Fahrzeug, Handbediengerät und das Sicherheitssystem. Zum externen Umfeld gehören dagegen die FTS-Leitsteuerung, andere FTFs, Arbeitsstationen und die Infrastruktur. Von einer globalen Materialflusssteuerung wird ein Transportauftrag erstellt und an die FTS-Steuerung übergeben. In der administrativen Ebene wird dieser verwaltet. Nach der Fahrzeugdisposition werden über die Schnittstelle zur operativen Ebene, Fahraufträge an die einzelnen Fahrzeuge übermittelt. In der operativen Ebene werden Fahraufträge durch den Manager abgearbeitet und bei Vollendung eine Rückmeldung an das überliegende System gemeldet. Der Manager koordiniert die Funktionen des Fahrzeuges und hat als zentrale Funktion die Auftragsverarbeitung und die Programmgenerierung. Dabei zerlegt der Manager den Fahrauftrag in Einzelbefehle. Diese werden als Fahr-, Energie-, Sicherheits- oder LAM-Befehle an die entsprechenden Funktionsmodule übergeben [8].

2.1.3 Navigation

Die Navigation spielt eine Schlüsselrolle im Bereich des FTS. Sie beeinflusst maßgebend die Infrastruktur und Flexibilität des Systems. Je nach Anwendungsfall werden unterschiedliche Navigationsverfahren, wie in Abbildung 2.6 dargestellt, angewendet [6], [9].

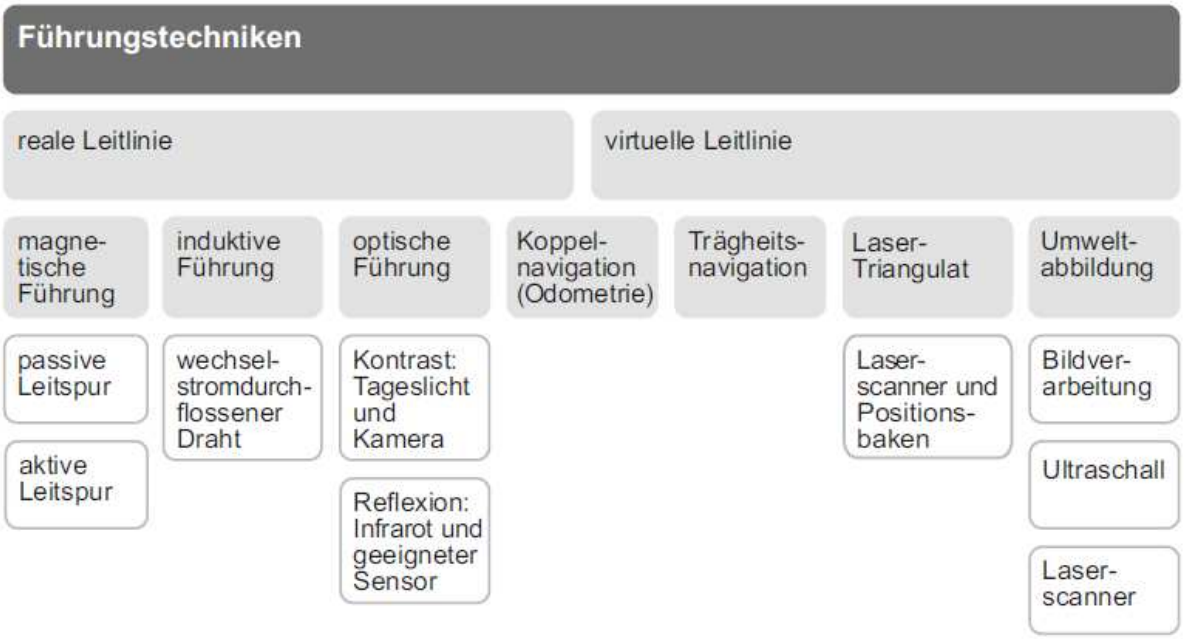

Abbildung 2.6: Führungstechniken eines FTS [10]

Koppelnavigation (Odometrie)

Bei der Koppelnavigation wird durch die bekannte Startposition und den zurückgelegten Weg die aktuelle Position des Fahrzeuges ermittelt. Der zurückgelegte Weg kann durch die Messung der Umdrehung eines Rades, deren Durchmesser bekannt ist und durch die Messung des Lenkwinkels bestimmt werden. Diese Methode allein ist jedoch recht ungenau, weshalb sie meist mit anderen Verfahren kombiniert wird. Die Messgenauigkeit hängt von der Bodenqualität, Verschmutzung bzw. Abnutzung der Räder und dem Schlupf des Motors ab [11].

Induktive Führung

Bei der induktiven Führung wird ein stromdurchflossener Leiter in den Boden eingelassen. Mit einem Frequenzgenerator wird Wechselstrom durch den Leiter geschickt, welches ein elektromagnetisches Wechselfeld hervorruft. Mithilfe zweier Spulen auf dem FTF wird durch das elektromagnetische Wechselfeld eine Spannung induziert. Durch die ermittelte Differenzspannung wird der Lenkmotor angesteuert [1], [12].

Magnetische Leitspur

Bei dieser Methode wird ein Metallstreifen auf dem Fußboden verklebt. Mithilfe von Magnetfeldsensoren unterhalb des Fahrzeuges wird die Feldänderung gemessen. Das Prinzip zur Regelung des Lenkmotors erfolgt simultan wie bei der induktiven Führung [11].

Optische Leitspur

Hier wird eine zur Umgebung kontrastierte Farbspur auf den Fußboden geklebt. Die Spurerkennung erfolgt mithilfe einer Kamera unter dem FTF und einer Bildverarbeitungssoftware, welche die Ansteuerungsgrößen an den Lenkmotor übermittelt [12].

Punktnavigation

Hierbei werden Bodenmarkierungen auf der Bahn angebracht, welche von der Sensorik des Fahrzeuges erfasst werden. Mögliche Positionsfehler von der Odometrie können anschließend an den Markierungen ausgeglichen werden. Als Referenzpunkte werden Quick Response (QR)-Codes , Radio Frequency Identification (RFID)-Transponder oder Magnete verwendet. Letztere müssen in den Boden eingelassen werden. Dementsprechend kann eine Fahrkursänderung recht kostenintensiv werden [9], [10].

Rasternavigation

Ähnlich der Punktnavigation wird in diesem Fall ein Raster aus Magneten, RFID-Transpondern oder Farbmarkierungen auf dem Boden installiert. Mithilfe der Transponder kann das FTF seine Position in Längs- und Querrichtung ändern. Der Vorteil gegenüber den physischen Leitlinien liegt in der Flexibilität der Fahrkursgestaltung. Nachteil ist der hohe Aufwand bei der Anbringung der Markierungen im Boden. Wie die Punktnavigation verfügt auch die Rasternavigation meist über eine zusätzliche Koppelnavigation [6], [9].

Lasertriangulation

Mit einem rotierenden Laserscanner werden Marken aus reflektierendem Material an den Wänden, Säulen oder Maschinen gescannt. Je nach Anforderung müssen mindestens zwei Marken gleichzeitig für die Positionsbestimmung lesbar sein. Die Auswertung der Software erfolgt über Echtzeit. Somit wird die Absolutposition (x, y, Winkel α) des Fahrzeugs innerhalb des ortsfesten Koordinatensystems der Halle während der Fahrt ermittelt. Die Lasernavigation ist recht flexibel, da der Fahrkurs bzw. Fahrweg virtuell existiert und mittels einer Software schnell geändert werden kann, ohne etwas an den Reflektormarken ändern zu müssen. Ein Nachteil ist, das die Reflektoren derart angebracht werden müssen, sodass sie störungsfrei arbeiten können [1].

Global Positioning System (GPS)-Aktiv sendende Marken

Die Ortung des Fahrzeuges erfolgt mittels Laufzeitmessung zu Satelliten oder zu stationären, codierten Radarreflektoren. Dieses Navigationsverfahren ist besonders für Anwendungen in sehr großen Räumen, für den Outdoorbereich bzw. bei Fahrten zwischen Gebäuden geeignet. Stationäre Systeme innerhalb von Gebäuden sind jedoch deutlich ungenauer als die aufwendigere Outdoor-GPS Methode [9].

Umgebungsnavigation / Bildverarbeitungsverfahren

In diesem Verfahren orientiert sich das Fahrzeug mittels Bildverarbeitungssystem an natürlichen Peilmarken, auf oder neben der Fahrbahn. Natürliche Peilmarken sind Landmarken wie Wände, Pfeiler, Türnischen etc. Die ortsfesten Peilmarken müssen dabei deutlich erkennbar sein. Mithilfe eines Laserscanners oder Ultraschallsensors vermisst das FTF seine Position laufend anhand der natürlichen Peilmarken. Diese Koordinaten werden mit dem auf der Fahrzeugsteuerung gespeicherten Karte verglichen. Eine Auswertsoftware berechnet anhand der Karte den Fahrweg. Zum Vermessen gehört die Technik SLAM (Simultaneous Localization and Mapping) zum Standard [1]. Die o. g. Navigationsverfahren sind in Abbildung 2.7 dargestellt.

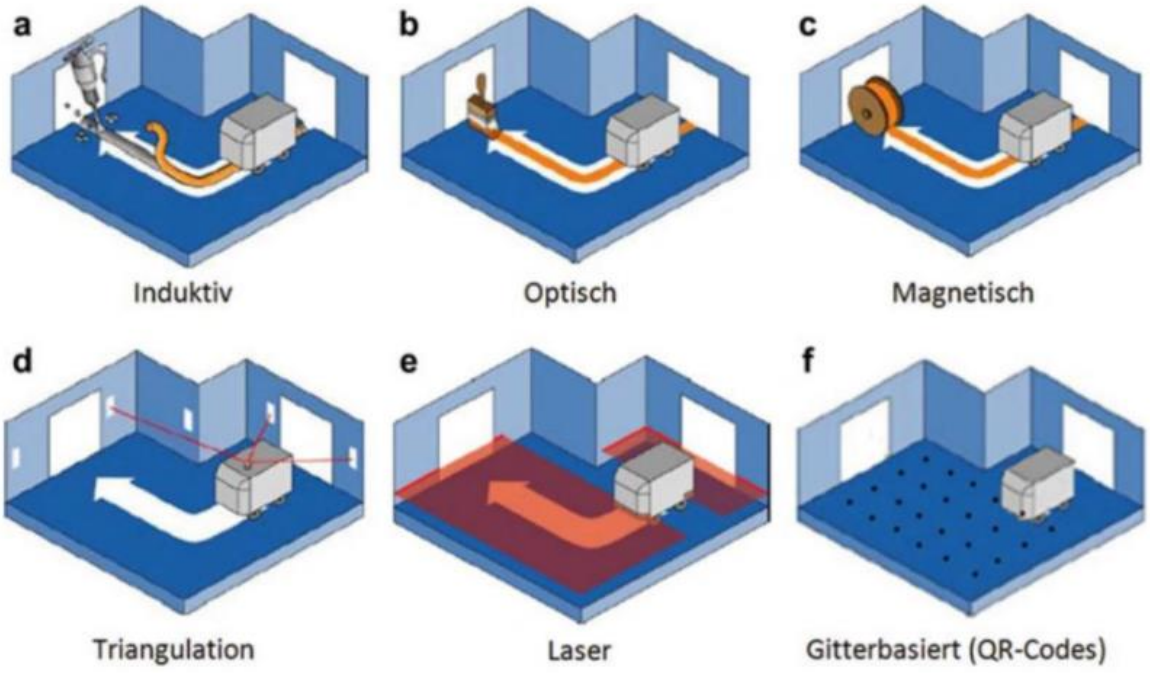

Abbildung 2.7: Navigationsverfahren für FTS; a.) induktive Führung, b.) optische Führung, c.) magnetische Führung, d.) Lasertriangulation, e.) Umgebungsnavigation mittels Laser, f.) Rasternavigation [13]

2.1.4 Infrastruktur und periphere Einrichtungen

Periphere Einrichtungen sind dadurch gekennzeichnet, dass sie speziell für den Einsatz von FTS aufgebaut und installiert werden. Eine Infrastruktur ist im Allgemeinen bereits im Anlagenumfeld vorhanden. Die Infrastruktur muss lediglich für den FTS-Betrieb ggf. angepasst und/oder erweitert werden. In Abbildung 2.8 sind die wesentlichen peripheren Einrichtungen und die Infrastruktur sowie deren Einbindung in das FTS dargestellt [14].

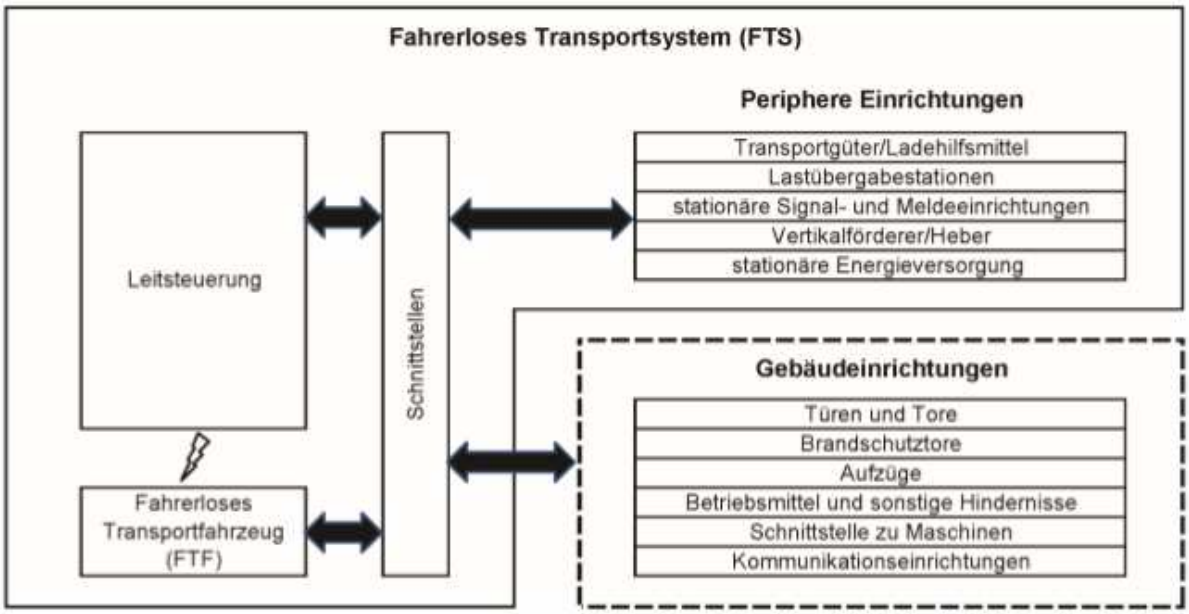

Abbildung 2.8: Infrastruktur und periphere Einrichtungen eines FTS [14]

Schnittstelle zum FTS-Netzwerk

Eine Kommunikation von Sensoren, Türen, Ruftaster, etc. mit der Leitsteuerung kann je nach Anwendung hinsichtlich physikalischer Schnittstelle und Schnittstellenprotokoll unterschiedlich ausgeführt werden. Die dafür verwendeten physikalischen Schnittstellen sind z.B.: Ethernet, Token Ring, RS 485, RS 422, RS 232 und Parallel-I/O. Aktuell verbreitete Protokolle sind z.B.: TCP/IP, H1-BUS, DUST 3964, Feldbusprotokolle wie Interbus, Profibus, CAN-Bus. In Abbildung 2.9 wird dargestellt, wie eine Kommunikation zwischen FTS und peripheren Einrichtungen über physikalischen Schnittstellen stattfinden kann [5].

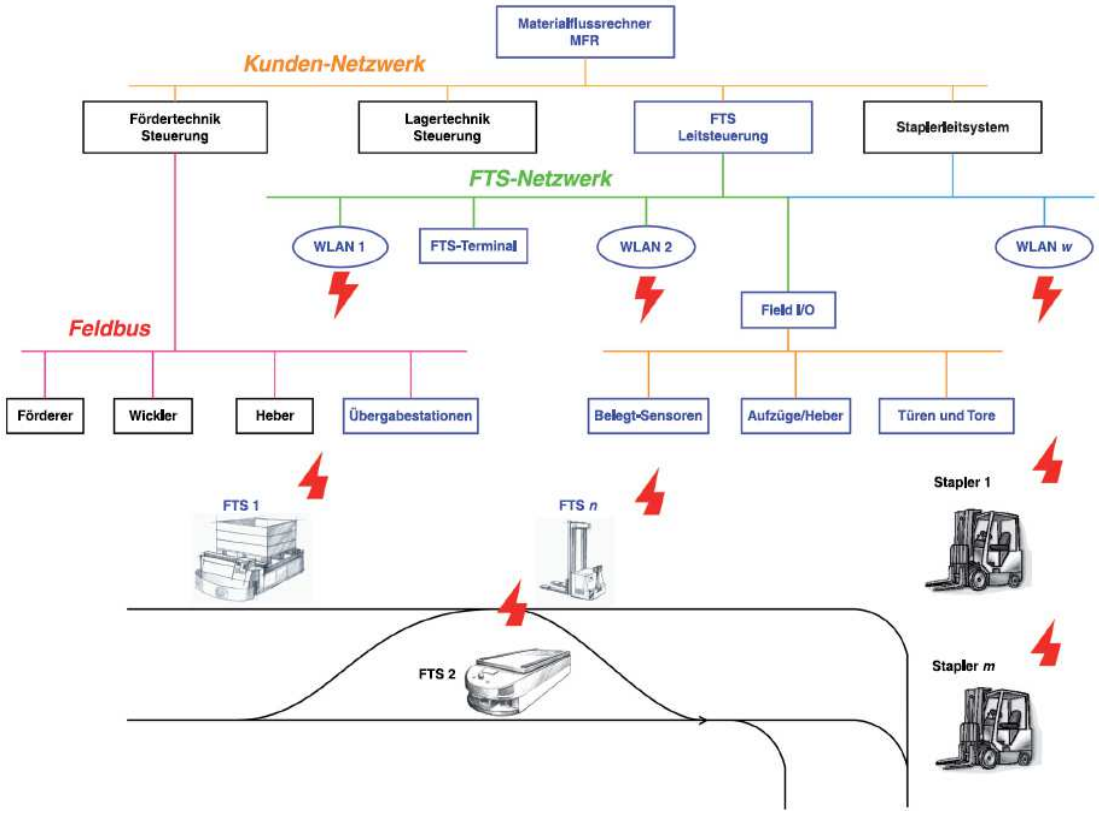

Abbildung 2.9: Schnittstellen eines FTS zu peripheren Einrichtungen [15]

Türen und Tore können automatisch von den FTF durchfahren werden, sofern sie über eine Öffnungs- und Schließautomatik verfügen. Hierbei bieten sich folgende Möglichkeiten an:

- Über eine LAN-Verbindung steuert die FTS-Leitsteuerung die Einrichtung
- Mittels Infrarot, Bluetooth oder WLAN kommuniziert das Fahrzeug direkt mit der Türsteuerung
- Die Einrichtung verfügt über eine eigene Sensorik, sodass ein sich näherndes Fahrzeug erkannt wird. Hierbei werden Sensoren verwendet wie u.a. Kontaktschleifen im Boden, Lichtschranken an den Wänden oder Bewegungsmelder [14].

In der Regel wird das Öffnungssignal so gesetzt, dass unabhängig von der Art des Tores die FTF ohne Verzögerung oder Stoppen diese durchfahren kann. Die Türen/Tore werden nach dem Passieren des Fahrzeuges mittels Signalaustausch geschlossen. Es muss sichergestellt werden, dass die Tür nicht schließt solange sich noch ein Fahrzeug in der Durchfahrt befindet und damit eingeklemmt werden kann. Für diese Fälle wird die Steuerung mit einer Zeitverzögerung versehen. Nach Auslösen eines Alarms muss das Fahrzeug innerhalb eines Zeitintervalls den Torbereich verlassen [1], [16].

2.1.5 Einrichtung zur Datenübertragung

Datenübertragungssysteme sorgen für ein informationstechnisches Zusammenwirken von stationärer Leitsteuerung, sonstigen stationären Einrichtungen und FTF. Berührungslose Datenübertragungstechniken eignen sich besonders für den Einsatz der FTS. Je nach Funktionsverteilung auf die Leitsteuerung bzw. die Fahrzeugsteuerung und der Vielfalt der Aufgaben der Fahrzeuge gestaltet sich die Kommunikation zwischen den stationären Steuerungen und den Fahrzeugen unterschiedlich. Der Datentransfer kann erfolgen zwischen [2]:

- stationärer Leitsteuerung und FTF
- ortsfesten Stationen und FTF
- FTF und FTF
- Leitsteuerung und ortsfesten Stationen

Ein wichtiger Aspekt bei der Datenübertragung ist der Kommunikationsinhalt. Dazu gehören u. a. Daten wie etwa der Fahrauftrag und die Statusmeldungen. Das Datenübertragungssystem muss in der Lage sein, die notwendigen Datenmengen ohne größere Verzögerung zu übermitteln. Die tatsächliche Leistungsfähigkeit eines Datenübertragungssystems ist abhängig von [2]:

- Übertragungsgeschwindigkeit
- Zuverlässigkeit des Verfahrens
- Prozedur und Protokoll der Datenübertragung

Datenübertragungssysteme

Eine Unterteilung des Datenaustausches erfolgt nach der Informationseinheit. Dabei wird ein Austausch zwischen binären Signalen und Datentelegrammen unterschieden. Binäre Signale werden durch einfache Elemente, wie z. B. Endschalter oder Lichtschranken oder durch komplexe Systeme durch Bündelung der Daten übertragen [2]. Tabelle 2 zeigt eine Einteilung der Datenübertragungsverfahren nach dem Wirkungsbereich.

Tabelle 2: Reichweite und Techniken der Datenübertragungsverfahren [2]

Datenübertragung		
punktbezogen	streckenbezogen	flächendeckend
Induktiv, Infrarot	Induktiv, Infrarot als Richtstrecke	Funk, Infrarot mit Weitwinkelcharakteristik

2.2 Dezentrale agentenbasierte FTS-Steuerung

Bis jetzt wurde bei der Erwähnung eins FTS immer von einem klassisch zentral gesteuerten Leitsystem gesprochen. Bei modernen FTS-Anlagen ist ein Trend in Richtung intelligenter, dezentraler Systeme zu beobachten [17]. Thanheiser et al. [18] definieren die Dezentralisierung als:

„*Verlagerung von Entscheidungskompetenzen innerhalb von Hierarchien - weg von zentralen Entscheidungsknoten hin zu einzelnen, lokal agierenden Handlungssubjekten*“

Bei dezentral gesteuerten FTS spielen Methoden aus dem Bereich der künstlichen Intelligenz (KI), die sog. Agententechnologie, eine wichtige Rolle. Die Fahrzeuge werden dabei als Agenten modelliert. Per Auktionsverfahren findet die Vergabe der Transportaufträge unter den Agenten statt. Die Datenverarbeitung findet dabei über einen zentralen Leitrechner statt. Für die Auftragsvergabe gibt jedes Fahrzeug ein Gebot ab, welches ein Kostenmaß darstellt. Auf dessen Grundlage wird entschieden, welches Fahrzeug für den Transportauftrag infrage kommt. Damit ein Fahrzeug ein Gebot abgeben kann, muss es über die passende Handhabungstechnik und über ausreichend Energie verfügen. Weiters werden die geplanten Zeiten für die Anfahrt zum Abholort und für den ggf. abzuarbeitenden Auftrag in das Gebot eingebunden [19].

Die Kommunikation der Fahrzeuge untereinander muss gewährleistet werden, damit möglichst viele Fahrzeuge an der Verhandlung teilnehmen können. Über eine WLAN-Infrastruktur erfolgt die Kommunikation zwischen den Fahrzeugen. Zukünftig wird auch die direkte Kommunikation D2D (Device to Device) mithilfe der 5G Technologie zwischen den Fahrzeugen möglich sein [20].

In Abstimmung mit anderen Fahrzeugen erfolgt das Koordinieren der Entscheidungen der Auktion eigenständig und unmittelbar untereinander. Fahrzeuge, welche neu hinzugefügt werden, melden sich eigenständig im System an und können dadurch an der Auktion teilnehmen. Die Entscheidung über die Auftragsvergabe wird durch die Agenten dezentral getroffen. Die Kommunikation nach Generierung des Auftrages erfolgt von der zentralen Einheit aus zu den Fahrzeugen [21].

Agententypen für eine AGV-Steuerung

Agenten, welche ähnliche Aufgaben übernehmen, können in Gruppen organisiert werden, welche als Agententypen zusammengefasst werden. Im Falle einer Anwendung auf FTS wurden von Schwarz [22], wie in Tabelle 3 dargestellt, die folgenden Agententypen definiert. Jede Entität ist dabei durch mehrere Agententypen komplett definiert. Für die Koordination der untergeordneten Agenten ist der Agententyp „*Manager*“ zuständig.

Tabelle 3: Agententypen für ein FTS nach Schwarz [22]

Bezeichnung	Funktion	
Manager	Steuerung und Koordination, Kommunikation mit anderen Entitäten (FTF, Stationen), Bereitstellung/Verarbeitung von Statusmeldungen	
RouteManager	Routenplanung, Verwaltung des Kartenmaterials	
TransportOrderManager	Auftragsabarbeitung	Annahme der Transportaufträge, Durchführung von Auktionen, Zerlegung der Transportaufträge in Fahraufträge
TravelOrderManager		Annahme der Fahraufträge, Zerlegung der Fahraufträge in Befehle und Übergabe an die zuständigen Agenten
SafetyManager	Schnittstellen zur Hardware	Sicherheitssystem
DriveManager		Bahnführung, Lagererfassung
PowerManager		Energiemanagement
LoadManager		Lasthandling

Zentrale Leitsteuerung vs. dezentrale Steuerung

Klassisch werden FTS zentral gesteuert. Dabei erhalten die Fahrzeuge lediglich die Fahrbefehle und Informationen über die auszuführenden Transportaufträge und führen diese aus. Die gesamte Koordinationsaufgabe wird dabei von der FTS-Leitsteuerung übernommen [1]. Aufgrund des Wunsches nach immer mehr kundenspezifischen individualisierten Produkten, werden anpassungsfähigere, flexiblere und intelligentere Fertigungssysteme gefordert, was wiederum eine hohe Herausforderung für die Intralogistik darstellt [23].

Feldmann und Wolfgang [24] weisen auf nicht mehr zufriedenstellende Lösungen mittels zentraler Steuerung der FTF hin, ausgelöst durch die Veränderungen der Produktionsbedingungen und -umgebungen. Durch die zentrale Steuerung wird keine Anbindung an das Geschehen vor Ort realisiert. Infolgedessen wird den Fahrzeugen ein unzureichender Entscheidungsraum gegeben. Stößt das Fahrzeug bspw. bei den auszuführenden Aufträgen auf Probleme, so müssen sie von der zentralen Instanz zur Problemlösung unterstützt werden. Darüber hinaus sprechen Sie über einen weiteren Nachteil der zentralen Steuerung, dem Single-Point-of-Failure. Ein Ausfall des zentralen Steuerungssystems, würde in diesem Fall zu einen Totalausfall des gesamten Transportsystems führen [24].

In einer dezentralen FTS-Steuerung koordinieren alle FTF die Aufträge selbst, indem sie durch Auftragsverhandlungen die auszuführenden Aufträge erhalten. Diese Aufgabe wird durch keine zentrale Instanz erledigt, weshalb die autonomen Entitäten, kooperierend und mit vorgegebenen Entscheidungswegen, dieser Aufgabe eigenständig nachgehen [25]. Varal [26] sieht die Einbringung einer dezentralen Steuerung für FTS als richtiger Weg in Richtung Industrie 4.0, mit der Begründung dass durch das Vorhaben der Industrie 4.0, welche eine Selbststeuerung der Produktion beabsichtigt, auch FTS betroffen sind und somit zumindest hinsichtlich ihrer Auftragssteuerung verändern müssen.

2.3 Grundlagen Produktionsleitsysteme

Um FTS in die IT-Architektur eines Unternehmens integrieren zu können, ist es äußerst notwendig mit übergeordneten Systemen wie etwa ERP oder MES zu kommunizieren. Rein zentral orientierte Systeme sind für Aufgaben mit derart hohem Komplexitätsgrad nicht geeignet. Stattdessen ist ein hierarchisch gegliedertes System vonnöten, welche die hohen Anforderungen hinsichtlich der Datenmenge und Geschwindigkeit erfüllt [27].

2.3.1 Ebenen-Modell eines Fertigungsunternehmens

Prinzipiell findet eine Unterteilung in drei Planungsebenen statt, welche da wären: Unternehmensleitebene, Fertigungsleitebene und Fertigungsebene. In Abbildung 2.10 wird der Aufbau eines Produktionsleitsystems eines produzierenden Unternehmens nach Wiendahl et al. [28] dargestellt. Die Ebenen sind dabei hierarchisch gegliedert. Ein weiterer Gesichtspunkt ist die Datenmenge und die Geschwindigkeit der Prozesse. Die zu handhabenden Dateninhalte werden immer kleiner und ihre Frequenz größer, je näher man an den Prozess kommt, während auf der anderen Seite die Anforderungen an Verfügbarkeit und an das Zeitverhalten rapide zunehmen [29].

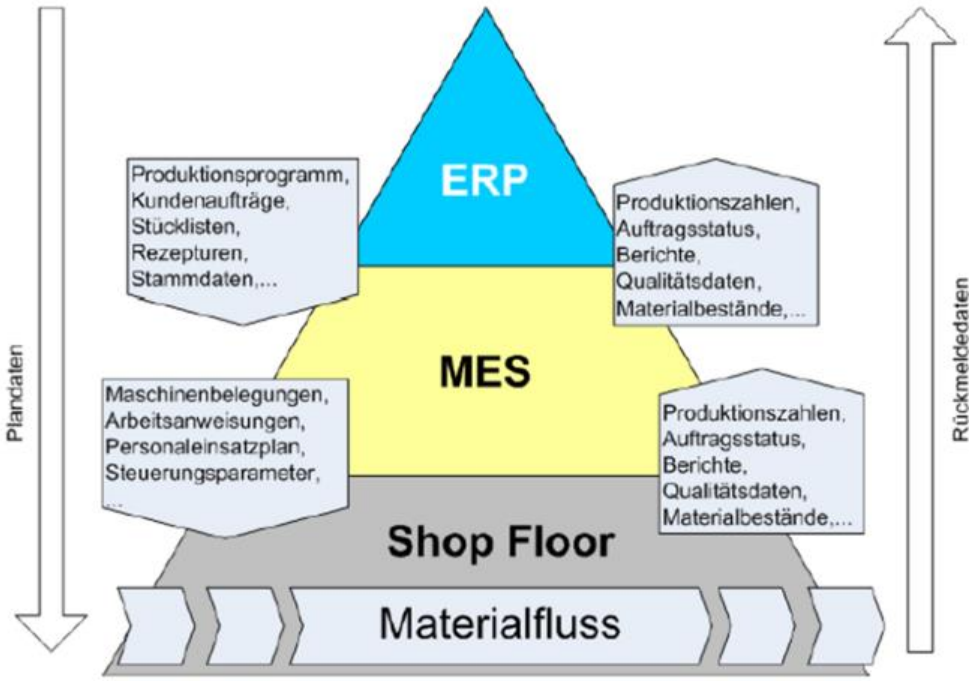

Abbildung 2.10: Ebenen-Modell eines Fertigungsunternehmens nach Wiendahl et al. [28]

ERP Systeme planen über einen längeren Zeitraum wie Wochen und Monate. Eine Feinplanung läuft in kürzeren Intervallen von Tagen und Schichten ab. Maschinen bzw. Anlagensteuerungen müssen hingegen innerhalb Minuten oder sogar Sekunden reagieren. Eine exakte Trennlinie kann zwischen den drei Ebenen nicht gezogen werden. Alleine durch die Funktion Datenerfassung ensteht eine enge Verknüpfung zwischen der Fertigungsleitebene und der Fertigungsebene. Zwischen der ERP- und MES-Ebene kann gesagt werden, dass MES mehr technologieorientiert ist, während ERP Systeme eher kommerziell angelegt sind [30]. Die grundsätzlichen Aufgaben der drei Planungsebenen in Bezug auf Aufträge, Ressourcen sowie Material sind in Tabelle 4 beschrieben.

Tabelle 4: Aufgaben der Planungsebenen nach der VDI-5600 [31]

	Unternehmensleitebene	**Fertigungsleitebene**	**Fertigungsebene**
	ERP	MES	Anlagen / Maschinen / Arbeitsplätze
Aufträge	Art und Umfang der Aufträge werden bestimmt, welche in einem ausgewählten Zeitraum zu produzieren sind. (Produktionsprogramm, Produktionsprogrammpla-nung)	Festlegung und Durchführung der Bearbeitungszeitpunkte. Zuordnung zu Ressourcen und Bestimmung der Reihenfolge für die Auftragsabwicklung im Produktionsprozess auf Ebene der Arbeitsgänge.	Umsetzung der Arbeitsgänge auf detailliertem Niveau, entsprechend den Vorgaben (Operationen, Schrittketten).
Ressourcen	Ressourcen werden nach kaufmännischen Aspekten verwaltet. Durchführung einer groben Kapazitätsbetrachtung.	Verwaltung der Ressourcen nach tatsächlichen Verfügbarkeiten und Zuständen. Ordnet Ressourcen für eine optimale Auftragsabwicklung.	Zur Durchführung der Arbeitsgänge werden die vorgegeben Ressourcen bedient.
Material	Ermittlung des Materialbedarfes und Auslösen von Auftragserzeugungen (oder Bestellvorgängen); Weiters werden die Bestände im Lager geführt.	Verwalten von Umlaufbeständen; der konkrete Materialeinsatz wird bestimmt und organisiert.	Das Material wird entsprechend der vorgegebenen Arbeitsgänge zu Zwischen- oder Endprodukten verarbeitet.

2.3.2 Enterprise Ressource Planning

Das ERP übernimmt die unternehmerischen Aufgaben, um die in einem Unternehmen vorhandenen Ressourcen (Kapital, Betriebsmittel oder Personal) möglichst effizient für den betrieblichen Ablauf einzusetzen und somit die Steuerung von Geschäftsprozessen zu optimieren [32]. Die Systemarchitektur eines ERP basiert auf dem Client-Server Prinzip. Hierbei werden Daten und Dienste von einem Server für Clients zur Verfügung gestellt [29].

Funktionsumfang von ERP Systemen

ERP Systeme können unterschiedliche Funktionsgruppen vorweisen. Der Funktionsumfang von ERP Systeme für kleine und mittlere Unternehmen sind im Vergleich zu Großunternehmen deutlich in ihrer Funktion begrenzt, da viele Funktionen wenig bis selten in Verwendung sind oder eine Erweiterung sich als zu kostenintensiv darstellt. Die wichtigsten Funktionsbereiche, die von einem ERP System abgedeckt werden, sind in der Abbildung 2.11 dargestellt. Des Öfteren werden Module oder Teilsysteme unterschiedlich bezeichnet und sind teilweise auch marketingabhängigen Wandlungen unterworfen [29].

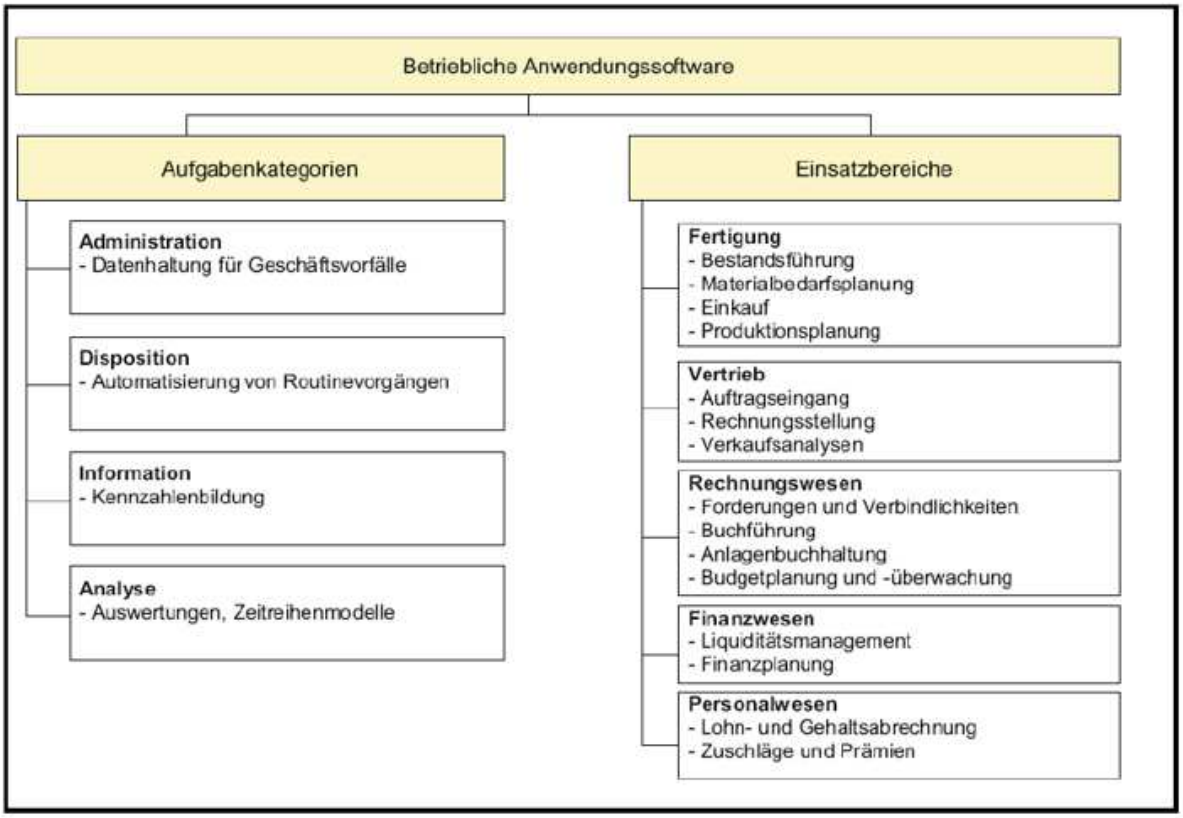

Abbildung 2.11: Aufgaben und Funktionen eines ERP Systems nach Gronau [33]

Produktion- und Fertigungssteuerung

Anstoß für betriebliche Planaufträge kommen aus Quellen wie Kundenbedarf/ Kundenaufträge, Prognosen und Vertriebsaufträge. Ein Auftrag kann dabei einzeln oder kombiniert aus den drei Bedürfnissen zusammengestellt werden. Die Reihenfolge und Terminierung werden über die Feinplanung gesteuert. Mithilfe der Fertigungsplanung werden die Anforderungen in Fertigungsaufträge umgesetzt. Basis dafür bilden die aus der Arbeitsplanung vordefinierten Arbeitspläne aus Stammdaten. Ein Arbeitsplan ist ein technologisches Dokument, welches vorgibt, wie ein bestimmtes Teil zu fertigen ist. Darin sind Maschinen, Werkzeuge, Reihenfolge der Arbeitsgänge und die Zeitvorgaben definiert. Diese Arbeitspläne bilden das Fundament und können je nach spezifischer Anforderung angepasst werden. In der Regel kommen Arbeitspläne nicht allein zum Einsatz, vielmehr fließen sie als ein Bestandteil in die gesamte Reihenfolgenplanung mit ein. Abbildung 2.12 zeigt den Ablauf zur Bearbeitung von Fertigungsaufträgen [29].

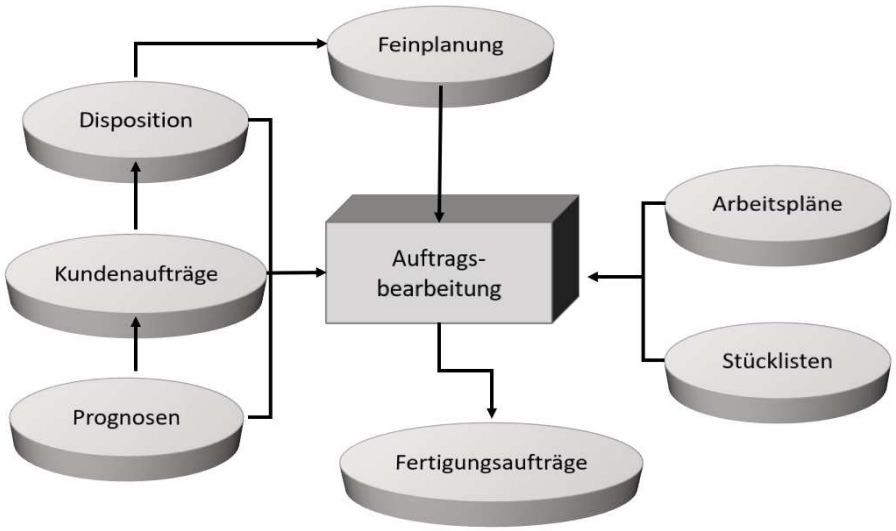

Abbildung 2.12: systematischer Ablauf von Fertigungsaufträgen nach Osterhage [34]

Dagegen beschäftigt sich die Fertigungssteuerung mit der Ausführung der Feinplanung in der betrieblichen Realität. Für die Durchführung werden spezielle Werkzeuge (Plattformen) benötigt, welche ein Teilpaket des gesamten ERP Systems darstellen oder als eigene Software auftreten, welche über Schnittstellen mit den Ebenen verbundenen werden. Diese Plattformen wie z. B. ein MES werden als Leitstand angegeben.

2.3.3 Manufacturing Execution System

Klassische ERP Systeme werden fälschlicherweise oft in der Aufgabe der Fertigungsoptimierung gesehen. Mittel- und langfristig gesehen arbeiten aber ERP Systeme mit einem zu geringen Detailierungsgrad. Ereignisse, welche in der Fertigung auftreten, wie etwa Unterbrechungen und Wiederaufnahme von Arbeitsgängen, Start/Stopp von Maschinen, etc., können vom ERP aufgrund unpassender Datencontainer und Verarbeitungsmechaniken nicht verarbeitet werden. Gleichzeitig sind die Daten der Anlagen und Prozesssteuerung für die Fertigungsorganisationen zu detailliert [31].

Das MES stellt dabei das Bindeglied zwischen der Ebene des Unternehmensmanagements und der Fertigungsebene dar. Plandaten, die ihren Ursprung in ERP Systemen haben, werden im MES verarbeitet und an die Shop-Floor Ebene weitergeleitet. Dadurch werden grob geplante Produktionsprogramme mithilfe der MES Funktion „*Feinplanung und Steuerung*“, anhand der aktuellen Ausgangslage im Shop Floor, geprüft. Infolgedessen werden u.a. mögliche Konflikte bei der Maschinenbelegung gelöst. Umgekehrt werden Rückmeldedaten aus der Automationsebene im MES erfasst. Diese Daten werden im MES verarbeitet und der Unternehmensleitebene in kompakter Form weitergeleitet [35].

Vertikale und horizontale Integration

Unter MES werden laut Kletti [36] heutzutage modular aufgebaute, integrierte Lösungen verstanden, die:

- Funktionsebenen des Unternehmensmanagements mit der Fertigung verbinden (vertikale Integration) und
- den gesamten Fertigungsprozess abdecken und damit bisherige Insellösungen ersetzen (horizontale Integration).

Die vertikale Integration sorgt für eine synchrone Zusammenarbeit eines Fertigungsunternehmens zwischen der Unternehmensebene und der Automatisationsebene. Voraussetzung dafür sind effektive Kommunikationswege. Die Daten müssen in den entsprechenden Ebenen verdichtet werden, sodass die unter- bzw. überlagerte Ebene mit brauchbaren Daten versorgt wird. In die Ebene des MES treffen dabei die großen Datenmengen aus der Fertigung ein. ERP Systeme bekommen aus dieser Datenmenge nur einen relativ kleinen Anteil zurückgemeldet [37].

Die horizontale Integration sorgt für eine Verknüpfung der Daten über alle am Fertigungsprozess beteiligten Ressourcen hinweg. Ziel dabei ist, autonome Insellösungen und zusätzliche Schnittstellen zu vermeiden. Ein integratives Datenmanagement des MES stellt sicher, dass alle Ressourcen wie Maschinen, Werkzeuge, Personal, NC-Programme oder Einstellparameter, Fertigungshilfsmittel, Prüfpläne sowie Prüfmittel

rechtzeitig verfügbar und optimal ausgelastet sind. Dies fördert eine Fertigung mit hoher Variantenvielfalt und flexibler Lieferfähigkeit [38]. In Abbildung 2.13 wird die vertikale und horizontale Integration eines MES nach Kletti [37] dargestellt.

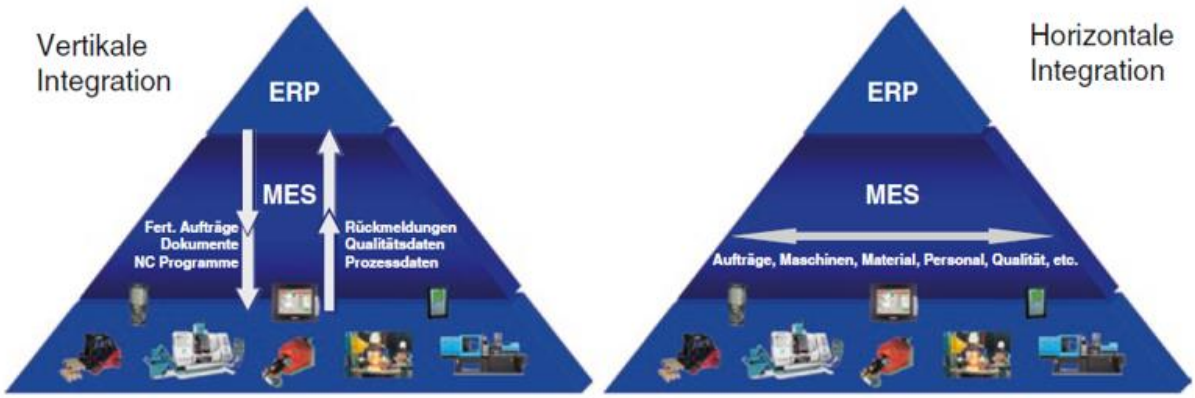

Abbildung 2.13: vertikale und horizontale Integration eines MES nach Kletti [37]

Funktionsgruppen eines MES

Je nach Autor können die Funktionsgruppen in der Benennung und Anzahl variieren. Ein MES muss dabei nicht jede Funktionsgruppe explizit umfassen. Der Leistungsumfang wird vielmehr den Anforderungen des Anwenders angepasst [31]. In der VDI Richtlinie 5600 wird zwischen zehn MES-Aufgaben unterschieden:

- **Auftragsmanagement**: Auslöser von Aktivitäten in einem Fertigungsunternehmen. Dient ebenso als Datencontainer für alle Informationen und Aufwände, die für die Abarbeitung notwendig sind bzw. dabei entstehen.
- **Feinplanung und -steuerung:** Hier werden Belegungspläne unter Berücksichtigung von Ressourcen- und Materialinformationen erstellt. Weiters sind diverse Regeln in Abhängigkeit der Produktionsziele oder des Arbeitsvorrats beinhaltet.
- **Betriebsmittelmanagement**: Beinhaltet die Termin- und bedarfsgerechte Verfügbarkeit der Betriebsmittel (Maschinen, Anlagen, Werkzeuge, etc.) und das Sichern der technischen Verfügbarkeit (Instandhaltung).
- **Materialmanagement**: Beinhaltet die termin- und bedarfsgerechte Ver- und Entsorgung der Fertigung mit Material, das Führen von Umlaufbeständen und Seriennummer-Verfolgung der Produkte und Halbzeuge.
- **Personalmanagement**: Beinhaltet das termingerechte Bereitstellen von Personal für den Produktionsprozess unter Berücksichtigung von Kapazitätsdaten.
- **Leistungsanalyse**: Beinhaltet das Bewerten und Analysieren der Prozesse mit dem Ziel, diese durch geeignete kurzfristige (Steuerung) und langfristige (Optimierung) Veränderungen zu regulieren.

- **Qualitätsmanagement**: Unterstützt das Qualitätsmanagement und die systematische Verbesserung der Produkt- und Prozessqualität.
- **Informationsmanagement**: Aufbereitung von Daten. Dient als Schaltstelle zur Integration anderer MES-Aufgaben und der Durchführung aller Workflows bei der Abarbeitung der Fertigungsprozesse und der Prozessoptimierung.
- **Energiemanagement**: Systematische Erfassung des Energiebedarfs eines Produktionsunternehmens. Liefert einen Beitrag zur energieeffizienten Fertigung im Unternehmen.
- **Datenerfassung**: Erfassen aller relevanten Daten aus den Produktionsbetrieben. Die Daten werden weiter geprüft und verdichtet.

Auftragsmanagement

Ein Auftrag ist der Auslöser einer Aktivität in der Produktion. Ebenso dient er als Datencontainer für alle Informationen und Aufwände, welche bei der Abarbeitung entstehen. Das Auftragsmanagement im MES darf nicht mit dem Auftragsmanagement in der ERP Ebene verwechselt werden. Die Behandlung eines Auftrages innerhalb des ERP Systems ist langfristig angelegt. Hingegen erfolgt im MES eine zeitnahe Behandlung der Aufträge. Weiters ist der Detailierungsgrad in beiden Ebenen recht unterschiedlich ausgeprägt [31]. Aufträge können dabei vom ERP eintreffen oder über externe Quellen übernommen werden. Diese Aufträge werden im Auftragsmanagement verwaltet und bei Bedarf mit zusätzlichen Informationen ergänzt, welche für die Abarbeitung der Aufträge notwendig sind. Nach Kurbel [29] können Aufträge mit folgenden Informationen ergänzt werden:

- Festlegung der Ziele als langfristige Vorgaben, welche sich aus dem kontinuierlichen Verbesserungsprozess ergeben (z. B. Vorgabe einer bestimmten Durchlaufzeit für eine Auftragsart)
- anwendungsspezifische Vorgaben, wie etwa Wartungszyklen einer Maschine, welche sich auf bestimmte Aufträge bezieht.
- Technische Regeln, Vorschriften und weitere Dokumentationen, wie z. B. Prüf- oder Arbeitsanweisungen, Grenzwerte, Plausibilitäts- und Verarbeitungsregeln, Skizzen, Abläufe, Maschineneinstellungen.

Daten und Informationen werden aus dem Fertigungsprozess den jeweiligen Aufträgen zugeordnet und werden in verdichteter Form rückgemeldet. In Abbildung 2.14 wird die Struktur der MES Aufgabe „Auftragsmanagement“ nach der VDI 5600 [31] dargestellt.

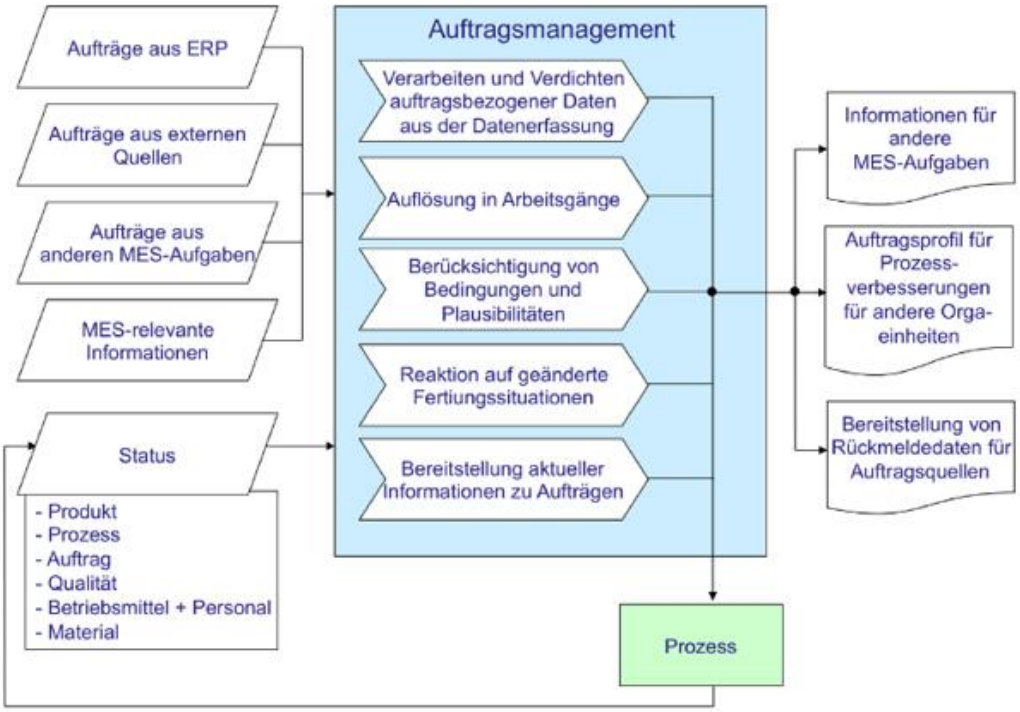

Abbildung 2.14: Bestandteile des Auftragsmanagement nach der VDI-5600 [31]

Auftrag/Arbeitsgang

Aufträge entstehen in der ERP Ebene und beschreiben das Soll für den Produktionsprozess über Planvorgaben. Parallel werden die Kosten für die Aufnahme von erbrachten Leistungen (Zeiten) und produzierter Menge aufgenommen. Somit sind Aufträge klassische Erfassungsobjekte. In der Umgebung der Datenerfassung wird unter einem Auftrag, meist der Arbeitsgang, die Arbeitsfolge oder der Vorgang verstanden. Der Arbeitsgang beinhaltet alle Informationen, die sich gemäß Arbeitsplan auf den jeweiligen Arbeitsplatz und die Prozessstufe innerhalb des Auftrags beziehen. Somit werden mithilfe des Arbeitsganges Fertigungsinformationen an den Erfassungsplatz transportiert. Zu diesen gehören Stammdaten aus dem Arbeitsplan (Vorgabegeschwindigkeit, Planarbeitsplatz, etc.), dem Materialstamm (Stücklisteninformation, Einsatzmengen, Zeichnungen) und zuletzt die beschreibenden Daten des Auftrages (Termin und Sollmenge, eventuell Kundeninformationen, Drucktexte für Etikettenlayouts etc.). Des Weiteren wird der Ressourcenbedarf mittels des Arbeitsganges bestimmt. Dieser wird der Fertigung implizit oder durch manuelle Eingabe als MES-Meldeobjekt zugeordnet. Gleichzeitig werden Informationen bzgl. des Fortschrittes der Fertigung aus dem Produktionsprozess erfasst und an die MES Datenbank übermittelt. Neben reinen Fertigungsaufträgen können auch Nacharbeitsaufträge, Projektaufträge, Gemeinkostenaufträge, Prüfaufträge, etc., verarbeitet werden [30].

2.4 Hypertext Transfer Protokoll

Das HTTP ist ein Client-Server Protokoll, welches festlegt, wie Nachrichten formatiert und übertragen werden und welche Aktionen, Webserver und Webbrowser als Reaktion auf verschiedene Befehle ausgeführt werden sollen. Wenn der Benutzer bspw. eine Uniform Resource Locator (URL) in den Browser eingibt, sendet der Browser einen HTTP-Befehl an den Webserver, welcher diese auffordert Daten abzurufen bzw. zu übertragen. Die grundlegenden Merkmale des HTTP sind laut Srinivasan [39]:

- HTTP verwendet das Request- / Response Paradigma, bei dem ein HTTP-Client eine Request Nachricht an den HTTP-Server sendet und der Server eine Response Antwort zurückgibt (siehe Abbildung 2.15)
- HTTP ist ein Pull-Protokoll, der Client ruft Informationen vom Server ab
- HTTP ist ein zustandsloses Protokoll, d. h. jeder Request-Response Austausch wird unabhängig behandelt. Clients und Server müssen keinen Status beibehalten. Ein HTTP Transaktion besteht aus einer einzelnen Anforderung von einem Client an einen Server, gefolgt von einer einzelnen Antwort vom Server zurück an den Client. Der Server verwaltet keine Informationen über die Transaktion. Bei einigen Transaktionen muss der Status beibehalten werden. Um einen Status aufrechtzuerhalten, verwendet die Website Proxys und Cookies
- HTTP ist medienunabhängig, d. h. jeder Datentyp kann per HTTP gesendet werden, wenn beide Seiten, Client und Server, wissen, wie sie mit dem Dateninhalt umgehen

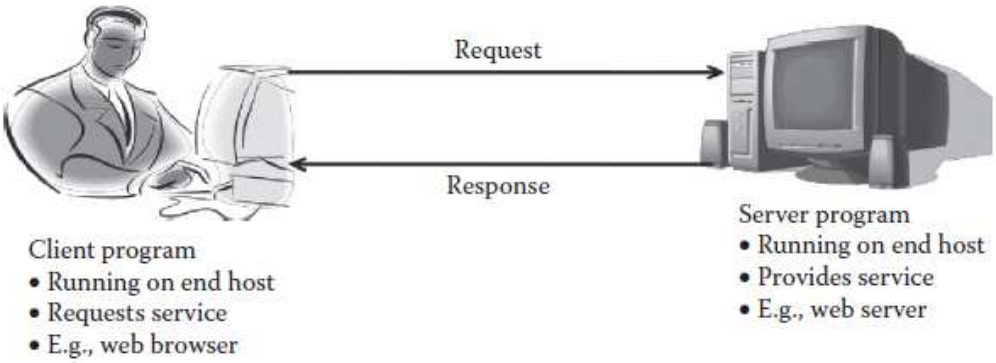

Abbildung 2.15: HTTP Request/Response Prinzip [25]

2.4.1 HTTP Request-Response Kommunikation

Der HTTP-Request setzt sich aus drei Hauptteilen zusammen, dargestellt in Abbildung 2.16. Der erste Teil bildet dabei die Anforderungen der Kommunikation. Hierbei wird zuerst die Request-Methode platziert und der genaue Pfad zu den gewünschten Ressourcen definiert. Am Ende der Befehlszeile wird noch die HTTP Version angeführt. In der zweiten Zeile folgen die HTTP-Headerfelder. Für eine funktionierende Anfrage ist die Angabe des Hosts zwingend erforderlich. Die nächste Zeile dient der zusätzlichen Übertragung von Parametern und Argumenten. Diese können u. a. die Kommunikation zwischen Server und Client wesentlich erleichtern. Weitere Daten können den Request beigefügt werden, diese müssen durch ein Leerzeichen vom Headerfelder getrennt sein [40], [41].

GET/index.html HTTP/1.1	Request Line	HTTP Request
Date: Fri, 16 Sep 2016 22:08:32 GMT Connection: Close	General headers	
User-Agent: Mozilla/5.0 Host: www.drakshikumar.com Accept: text/html, text/plain Accept-Language: en-US Content- Length: 35	Request header	
BLANK LINE		
	Message Body	

Abbildung 2.16: Beispiel für eine HTTP-Request Nachricht [25]

Eine Response ist die Antwort des Servers auf eine Anfrage des Clients. Das Format ist nur geringfügig anders als beim Request. Auch hier besteht der Code aus drei Hauptteilen (siehe Abbildung 2.17). Die erste Zeile gibt die HTTP-Version und den Statuscode wieder. Die Statuszeile teilt dem Client mit, welche HTTP-Version verwendet wird und fasst die Ergebnisse der Clientanfrage zusammen. In der Header Zeile werden, analog zum Request-Headerfeldern, Parameter und Argumente übermittelt. Die übertragenen Daten stehen nach den Header-Informationen und werden durch ein Leerzeichen von diesen getrennt [40].

HTTP/1.1 200 OK	Status Line	HTTP Response
Date: Fri, 16 Sep 2016 22:08:32 GMT Connection: Close	General headers	
Server: Apache/2.4.7 Accept-Ranges: bytes	Response header	
Content-Type: text/html Content- Length: 95 Last-Modified: Mon, 5 Sept 2016 10:14:24 GMT	Entity header	
BLANK LINE		
<html> <head> <title> Welcome to my Homepage </title> </head> <body> Coming Soon! </body> </html>	Message Body	

Abbildung 2.17: Beispiel für ein HTTP-Response-Nachricht [25]

2.4.2 HTTP Methoden

Die Request Methode gibt den Typ der Anfrage an, die ein Client sendet. Eine Methode macht eine Nachricht entweder zu einer Anfrage oder zu einem Befehl. Ein Befehl weist den Server an, eine bestimmte Aufgabe auszuführen. Folgend werden die wichtigsten Request-Methoden aufgelistet und deren Funktion erläutert [39], [41]):

- **GET:** fordert eine Ressource von einem Server per URL an. Die URL in der Anforderungszeile identifiziert die Ressource. Wenn sie gültig ist, liest der Server den Inhalt der Ressource und sendet den Inhalt an den Client zurück, andernfalls wird eine Fehlermeldung an den Client zurückgesendet.
- **POST:** Senden eines Datensatzes von einem Client an den Server. Die tatsächlichen Informationen sind im Hauptteil der Request-Nachricht enthalten. Da die Daten im Text enthalten sind, können große Datenblöcke zum Server gesendet werden. Folglich kann die Länge der Anfrage unbegrenzt sein.
- **PUT:** bei dieser Methode werden dem Server Daten gesendet und er wird aufgefordert, diese zu speichern. Sie wird hauptsächlich verwendet, um neue Ressource hochzuladen oder ein vorhandenes Dokument zu ersetzen.
- **DELETE:** fordert den Server auf, die im URL beschriebene Ressource zu löschen.
- **TRACE:** fordert den Server auf, eine Kopie der vollständigen HTTP Request-Nachricht zurückzugeben, einschließlich Startzeile, Headerfelder und Text, die vom Server empfangen werden.
- **CONNECT:** wird verwendet, um eine Anforderungsverbindung in einen transparenten TCP/IP-Tunnel zu konvertieren. Dies wird normalerweise durchgeführt, um die Verschlüsselung der gesicherten Secure Socket Layer (SSL) zu erleichtern.
- **OPTIONS:** fordert den Server auf, eine Liste der möglichen verwendbaren HTTP-Methoden zurückzugeben, um auf die vom Request-URL angegebene Ressource zuzugreifen.
- **COPY**: kann verwendet werden, um eine Datei von einem Speicherort zu einem anderen zu Kopieren.

2.4.3 HTTP Status-Codes

Die Statuszeile besteht aus drei Teilen: HTTP-Version, Statuscode und Statusphrase. Zwei aufeinanderfolgende Teile sind durch ein Leerzeichen getrennt. Die HTTP-Version gibt die Version des Servers an (aktuell HTTP/1.1). Der Statuscode ist ein dreistelliger Code, der den Status der Antwort angibt. Für Maschinen ist der Status-Code einfach zu lesen. Um den User diese Codes verständlich rüberzubringen, wird u. a. eine Nachricht mit beigefügt. Die Status Codes werden hinsichtlich ihrer Funktionalität, wie in Tabelle 5 dargestellt, in fünf Gruppen eingeteilt [39].

Tabelle 5: HTTP-Status Codes [39], [41]

Code	Bedeutung
1xx	Informationen: wie etwa „Anfrage erhalten“ oder Fortsetzung des Prozesses. Wird häufig angezeigt, wenn der Client die POST- oder PUT-Methode verwenden will, um eine große Datenmenge auf dem Server zu speichern. Der Client überprüft dabei ob der Server die Daten verwalten kann, anstatt alle Daten auf einmal zu senden.
2xx	erfolgreiche Operation: diese Klasse zeigt die Anforderungen des Clients an, welche erfolgreich empfangen, verstanden und akzeptiert wurden.
3xx	Umleitung: gibt an das weitere Maßnahmen erforderlich sind, um die Anforderung abzuschließen.
4xx	Client-Fehler: diese Statuscodes werden verwendet, um anzuzeigen, dass der Client bei der Anfrage einen Fehler verursacht hat und somit die Anfrage nicht erfüllen kann.
5xx	Server-Fehler: diese Statuscodes zeigen an, dass beim Server ein Problem aufgetreten ist und somit die Anfrage derzeit nicht ausgeführt werden kann.

2.5 Message Queuing Telemetry Transport

Das MQTT ist ein Client-Server Transferprotokoll. Neben HTTP gehört MQTT zu den wichtigsten Protokollen im Bereich Internet of Things (IoT). Das Konzept basiert auf einem Publisher/Subscriber Kommunikationsprotokoll, welches eine asynchrone Nachrichten-basierte Übertragung zwischen Applikationen ermöglicht (siehe Abbildung 2.18). Dieses Prinzip macht das MQTT ideal für die "Machine-to-Machine" (M2M)

Kommunikation und speziell für die Anwendung von asynchroner one-to-many, many-to-one und many-to-many Kommunikation [42]. Der Inhalt einer MQTT Nachricht wird als Payload bezeichnet. Dabei können beliebige Daten wie Zahlenwerte, Texte (Strings) oder komplexere Datenstrukturen wie JSON übertragen werden. Der Datentyp und die Formatierung des Nachrichteninhaltes auf Sender- und Empfängerseite müssen dabei bekannt sein [43].

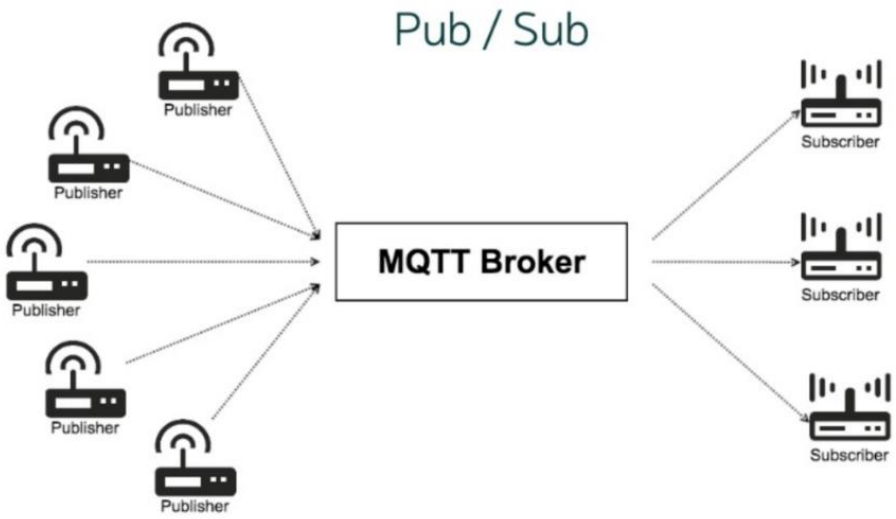

Abbildung 2.18: MQTT Publish/Subscribe Prinzip [43]

Client

Clients können sowohl Publisher als auch Subscriber sein. Prinzipiell ist es im MQTT möglich, dass ein Client beides zur selben Zeit ist. Clients können Microcontroller oder auch jedes andere Gerät, wie ein Server, sein. Dieses Gerät muss über ein Netzwerk mit dem Broker verbunden sein [44].

Broker

Der Broker ist das Kernstück einer jeden Kommunikation mit MQTT. Diese hat als Aufgabe, Nachrichten zwischen den einzelnen Applikationen zu verteilen. Die Nachrichten werden dabei von einem Publisher empfangen, nach Bedarf gefiltert und an die Subscriber weitergeleitet. Ein Broker kann dabei mit mehreren Clients verbunden werden. Die Nachrichten werden nach dem Push-Prinzip verteilt, d. h. Geräte müssen beim Broker nicht nach aktuellen Daten anfragen [45].

2.5.1 Topic-Adressierung

Mithilfe von Topics wird der Nachrichtenaustausch zwischen Client und Broker, bzw. dessen Adressierung organisiert. Die Topics werden vom Publisher festgelegt, auf denen sie ihre Nachrichten dem Broker übermitteln. Der Broker sammelt diese Topics und verteilt sie an die Subscriber, welche ein oder mehrere Topics abonniert haben. Somit wird dem Broker mitgeteilt, welche Subscriber welche Topics besitzen und über welchen Publisher die Nachricht eingeht. Topics können aus mehreren Ebenen bestehen und werden durch das Zeichen „/“ voneinander getrennt [46]. Ein Beispiel für die Verwendung von Topics ist in Abbildung 2.19 dargestellt.

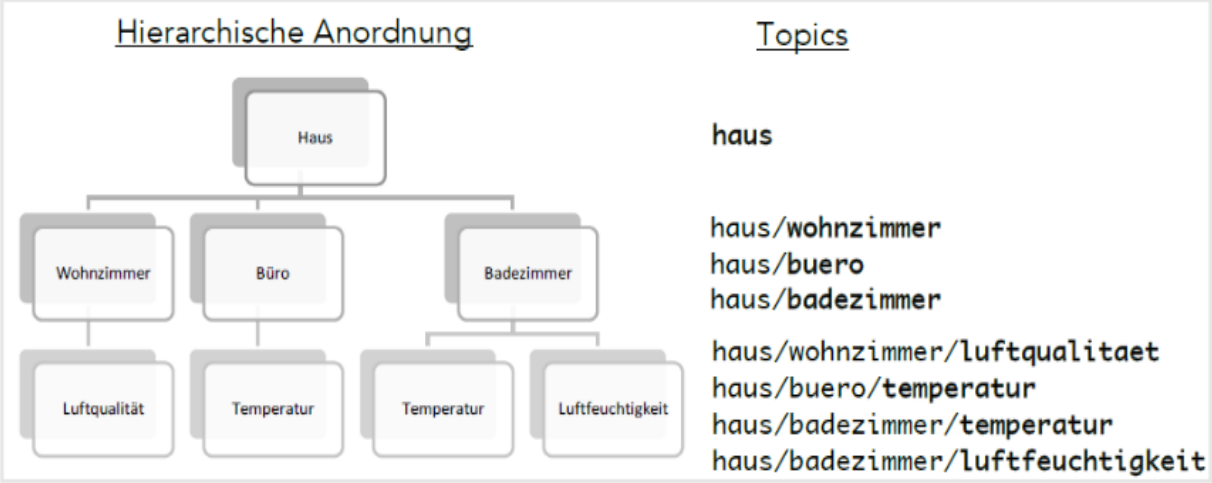

Abbildung 2.19: Beispiel für Aufbau und hierarchische Anordnung eines Topics [46]

Bei der Benutzung von Topics können zusätzlich sog. Wildcards verwendet werden. Dabei ersetzt eine Wildcard einen bestimmten Teil des Topics. Somit erhält ein Subscriber ggf. Nachrichten aus mehreren Topics. Wildcards werden dabei in Single Level Wildcards (Symbol: +) und Muti Level Wildcards (Symbol: #), unterschieden [47].

Eine weitere Funktion, welche von MQTT angeboten wird, ist der „*Last Will*“. Hier kann ein Subscriber dem MQTT Broker beim Herstellen der Verbindung seinen letzten Willen mitteilen. Es handelt sich dabei um ein Topic mit entsprechender Nachricht, die beim Abbruch der Verbindung mit dem Gerät veröffentlicht wird [33].

2.5.2 Quality of Service - QoS

Eine weitere Charakteristik vom MQTT ist die QoS. Diese gibt das Sicherheitsniveau für ankommende Nachrichten an. Dabei gibt es drei Level des QoS, welche in Tabelle 6 gelistet sind [46].

Tabelle 6: QoS-Level (vgl. [34])

QoS Level	Zustellung	Zustellgarantie
0	maximal 1-mal	keine Garantie
1	mindestens 1-mal	Garantie der Zustellung, Duplikate möglich
2	genau 1-mal	Garantierte Zustellung, keine Duplikate

QoS 0: die Nachricht wird maximal einmal zugestellt und wird danach verworfen. Die Nachricht kommt dann entweder einmal an oder gar nicht. Infolgedessen wird die Nachricht auch kein zweites Mal gesendet.

QoS 1: Es wird garantiert, dass die Nachricht zumindest einmal beim Empfänger ankommt. Unter Umständen kann die Nachricht auch mehrmals beim Empfänger ankommen und als Duplikat auftreten.

QoS 2: die Nachricht wird genau einmal zugestellt. Es wird garantiert das die Nachricht maximal einmal beim Empfänger ankommt [48].

2.5.3 JavaScript Objekt Notation

JSON ist ein schlankes Format basierend auf den Datentypen der Programmiersprache JavaScript. Aufgrund seiner Einfachheit und der Tatsache, dass es sowohl von Maschinen als auch von Menschen leicht zu lesen ist, wurde JSON schnell eines der beliebtesten Formate für den Datenaustausch im Internet [49]. Vereinfacht gesagt ist JSON eine Textdarstellung, die durch bestimmte Regeln die Struktur der Daten definiert. Dabei baut die Zusammensetzung der Daten auf den folgende zwei Strukturen auf [50]:

- eine Sammlung aus Property-Value Paaren. Wird in verschiedenen Sprachen als Objekt, Datensatz, Dictionary, Array, etc. realisiert.
- eine geordnete Liste von Werten. Diese werden häufig als Array, Vektor, Liste oder Sequenz verwirklicht

Ein Objekt besteht aus einer ungeordneten Menge von Property-Value Paaren (siehe Abbildung 2.20). Es beginnt immer mit einer geöffneten geschwungenen Klammer und endet mit einer geschlossenen geschwungenen Klammer. Der obere Pfad der Abbildung beschreibt ein leeres Objekt. Im mittleren Pfad enthält das Objekt genau einen Wert. Der untere Pfad zeigt, dass ein Objekt aus einer beliebigen Anzahl von Zeichenfolgen- / Wertepaaren bestehen kann, die voneinander durch ein Komma getrennt werden [51].

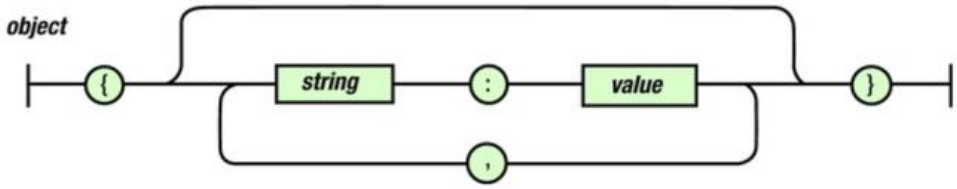

Abbildung 2.20: Syntax eines Strings-Value Paares [51]

In Arrays findet eine geordnete Sammlung von Werten statt (siehe Abbildung 2.21). Ein Array beginnt dabei mit einer offenen eckigen Klammer und endet mit einer eckigen geschlossenen Klammer. Analog zu einem Objekt kann das Array auch leer sein. Beinhaltet das Array mehrere Werte, werden diese ebenfalls durch ein Komma getrennt [51].

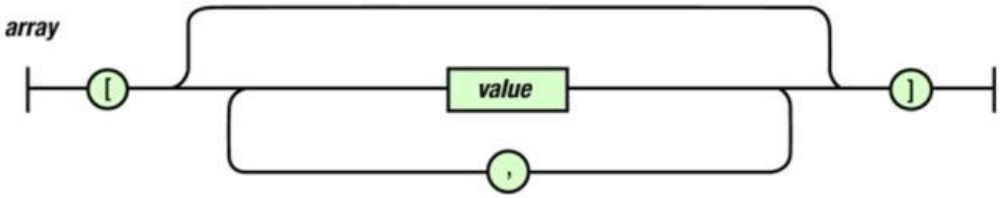

Abbildung 2.21: Syntax-Diagramm einer Liste [51]

Ein Wert kann dabei verschiedene Datentypen annehmen (siehe Abbildung 2.22). Dabei werden Strings in doppelte Anführungszeichen eingeschlossen [51].

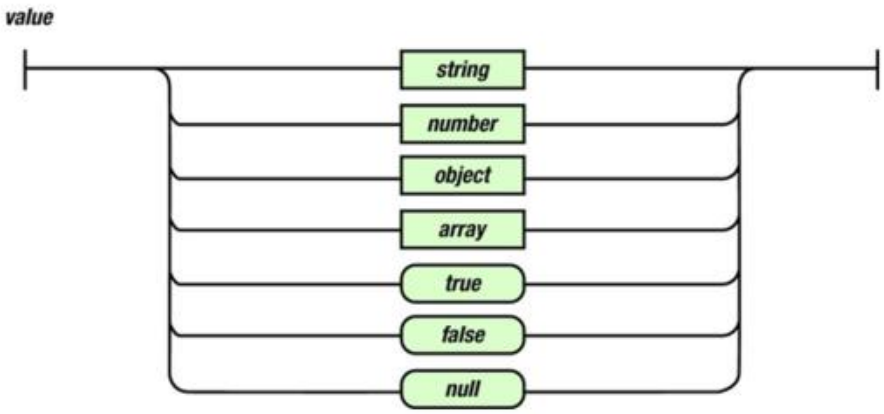

Abbildung 2.22: Syntax-Diagramm zur Veranschaulichung möglicher Datentypen [51]

3 Funktionen und Schnittstellen eines Flottenmanagers

In diesen Abschnitt werden zuerst die Hauptfunktionen eines Flottenmanagers behandelt. Weiters werden Aspekte für die Implementation der Software in eine MES-Umgebung präsentiert und Möglichkeiten der Realisation einer Schnittstelle zwischen Flottenmanager und Fahrzeugen behandelt. Im Anschluss werden auf dem Markt verfügbaren Software-Lösungen vorgestellt, welche sich für einen Einsatz in der TU Pilotfabrik eignen.

3.1 Funktionen eines Flottenmanagers

Ein Flottenmanager wird üblicherweise zur Steuerung und Verwaltung einer Flotte von FTF in industriellen Umgebungen eingesetzt. Sie zielen hauptsächlich darauf ab, die Transportzeit von Teilen und Produkten durch Optimierung der Routen und einer richtigen Zuordnung von FTF für eine Aufgabe zu verkürzen [52].

Wie schon im Kapitel 2.1.1. erwähnt, sind die Funktionen „Transportauftragsabwicklung“ und „Benutzer-interface“ essenziell für die Leitsteuerung. Damit mehrere Fahrzeuge konfliktfrei arbeiten können, sind zum Steuern und Verwalten der Aufträge gewisse Funktionen und Algorithmen nötig, zu diesen zählen:

- Routing
- Scheduling/Planung
- Fahrzeugdisposition

3.1.1 Routing

Ein Fahrbefehl an ein FTF besteht in seiner Struktur aus einem Graphen, welcher wiederum aus Knoten und Kanten besteht. Knoten sind Punkte, über die das FTF z. B. eine Positionsmeldung an die Leitsteuerung übermittelt. Anhand dieser kann der Fortschritt der Auftragsbearbeitung nachvollzogen werden. Die Verbindungen zwischen den Knoten werden als Kanten bezeichnet [53].

Durch ein erstelltes oder importiertes Führungswegnetzwerk kennt die Leitsteuerung alle Knoten und Kanten, um einen Zielpunkt zu erreichen. Auf dieser Grundlage berechnet die Leitsteuerung die ideale Route und übermittelt dem FTF ausschließlich diese Kanten und Knoten. Das Fahrzeug fährt anschließend im Rahmen der mechanischen Möglichkeiten die Knoten optimal ab. In Abbildung 3.1 wird der Unterschied der Routing-Informationen zwischen der Leitsteuerung und des FTFs dargestellt [53].

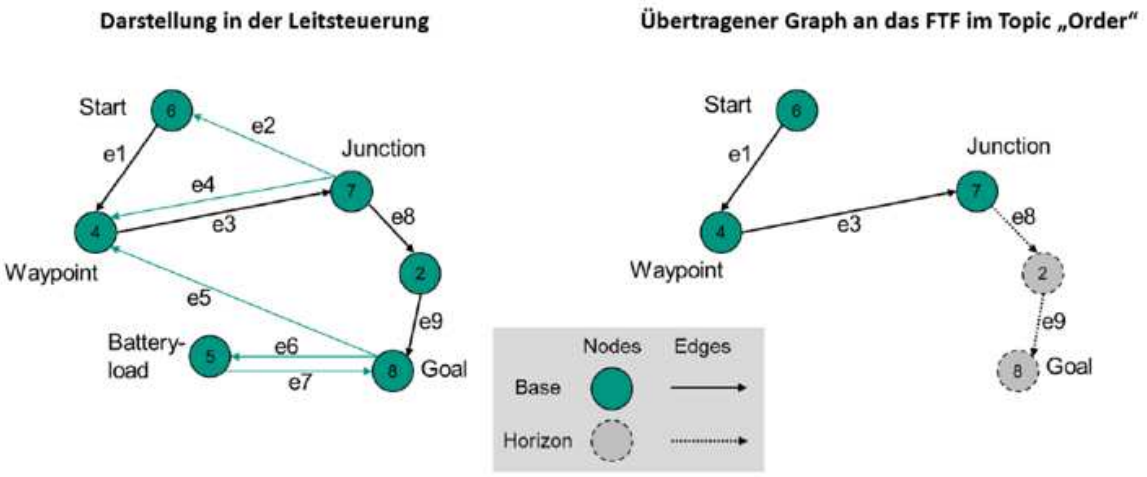

Abbildung 3.1: Unterschied der Routing-Information zwischen Leitsteuerung und Fahrzeug [53]

Hinsichtlich der Berechnung der idealen Route haben sich zwei Strategien etabliert, einerseits das „statische Routing“ und andererseits das „dynamische Routing“. Erfolgt das Routing statisch, ist der ermittelte Pfad, welcher das AGV zwischen den Knoten durchfährt, immer derselbe. Dies bedeutet, dass sich die Route im Laufe der Zeit als Reaktion auf die Umgebung, z. B. durch aktuelle Überlastung im System, nicht verändert. Dieser rein deterministische Ansatz ist in der Praxis weit verbreitet. Statisches Routing hat den Vorteil, dass es einfach, klar und leicht zu implementieren ist. Ein Hauptnachteil des statisch deterministischen Fahrzeugrouting ist es, dass das Routing nicht auf Änderungen in der Umgebung, wie Blockaden und Fahrzeugpannen, reagiert. Somit kommt es vermehrt bei überlasteten Systemen zu Blockaden, den sog. „Deadlocks“. Darüber hinaus kann das Routing unnötig lang werden, was gleichbedeutend mit einer erhöhten Transportzeit ist und somit zu einem geringen Durchsatz führt [54], [55].

Beim dynamischen Routing kann das AGV unterschiedliche Wege einschlagen. Durch die Kommunikationsverbindung zwischen Router und dem Fahrzeug kann eine vorausschauende Planung der Route durchgeführt werden. Die Route kann während der Ausführung des Auftrages geändert werden. Somit ist es möglich, bei Streckenausfällen, Hindernissen und Überschneidungen mit Routen anderer Fahrzeuge eine alternative Route zu ermitteln und an das Fahrzeug zu übergeben. Dadurch können blockierte Knoten und Pfade umfahren werden, um mögliche Deadlocks zu vermeiden. Hinsichtlich der Effizienz zeigt Sahib [55], dass ein dynamisches Routing einer statischen Fahrzeugroutingstrategie überlegen ist.

3.1.2 Scheduling/Planung

Ziel der Planung von AGV's ist es eine Reihe von Fahrzeugen zu versenden, um Arbeitsprozesse wie das Be- und Entladen von TS, unter Berücksichtigung bestimmter Bedingungen (z. B. kürzeste Bearbeitungszeit, geringste AGV-Leerlaufzeit, usw.) erfolgreich durchzuführen [56].

Nach Le-Ahn [57] besteht das Hauptziel der meisten Planungsprobleme darin, Transportlasten innerhalb eines eingeschränkten Zeitfenster so schnell wie möglich zu transportieren. Dadurch werden andere Parameter wie die Wartezeit von beladenen Fahrzeugen und die Anzahl von kritischen Transportaufträgen in Warteschlangen reduziert. Die Planung beinhaltet im Wesentlichen zwei Aufgaben. Nämlich die Ermittlung der geplanten Start- und Ausführungszeit von Arbeitssätzen und einem Interaktionsmechanismus, welcher den Fortschritt des Zeitplans überwacht und mit unvorhergesehenen Aktionen (Stornierungen, Datumsänderungen, Pannen) umgeht. In der Praxis wird die Planung mit zwei klassischen Szenarios konfrontiert. Das erste Szenario beschäftigt sich mit der Erfüllung aller Prozesse unter Einschränkung von vorgegebenen Prioritäten. In Szenario zwei geht es hauptsächlich darum, die Planung so zu verbessern, dass die gesamte Fahrzeit aller Fahrzeuge reduziert wird oder die Anzahl der benötigten Fahrzeuge bei gleichem Output minimiert wird [55].

3.1.3 Fahrzeugdisposition

Mittels der Fahrzeugdisposition wird das „günstigste" Fahrzeug für den jeweiligen Transportauftrag von der Transportauftragsverwaltung ermittelt. Nach der Ermittlung des geeigneten Fahrzeuges wird ein Fahrauftrag an die Fahrauftragsabwicklung übergeben. Bezüglich der Ermittlung des günstigsten Fahrzeuges gibt es in der Literatur etliche Strategien, die von dem Führungspfadnetzwerk bis auf das jeweilige Anwendungsgebiet angepasst sind. Im einfachsten Fall wird das Fahrzeug ausgewählt, welches zum aktuellen Zeitpunkt verfügbar ist. Komplexe Systeme erfordern jedoch mehr Aufwand. Hierzu existieren Fahrzeugdispositionsstrategien wie etwa [1]:

- kürzeste Distanz zur Quelle, weg- oder zeit-optimiert
- Aufnahme mehrerer Lasten an unterschiedlichen Quellen
- unterschiedliche Fahrzeugtypen im System
- Prognosen über den zukünftigen Systemzustand

Für diese Strategien bilden die Knoten und Kanten aus dem Routing die Basis für die Berechnung. In der Fahrzeugdisposition werden noch zusätzliche Aufgaben, wie Batterielademanagement und der Umgang mit Leerfahrzeugen bewältigt. Beim Handling mit Leerfahrzeugen kann ein Fahrzeug, welches in absehbarer Zeit keinen Auftrag bekommt, entweder zu einer Dockingstation geleitet oder in einer Schleife bewegt, um im Falle des Bedarfs schneller einsatzfähig zu sein [1].

3.2 Schnittstelle zu übergeordneten Systemen

Eine optimale Lösung zur Steuerung eines FTS ist die direkte Anbindung der Leitsteuerung an ein ERP bzw. MES. Dadurch lassen sich alle Prozesse, Fahrrouten sowie Routenentscheidungskriterien transparent und nachverfolgbar in einem System abbilden. Das übergeordnete System generiert anhand der Produktionsvorgaben und weiteren Entscheidungskriterien die richtigen Ziele für die Fahrbefehle und schickt diese Informationen direkt an die Leitsteuerung. Diese wiederum verarbeitet die Daten und sendet die Informationen via WLAN an die FTF weiter. Dank integrierter Schnittstellen, sowohl vom übergeordneten System als auch von der Leitsteuerung, ist eine solche Einbindung direkt und vollkommen ohne den Einsatz einer zusätzlichen Middleware möglich. Abbildung 3.2 zeigt die Einordnung des Flottenmanagers im Ebenen-Modell nach Munz et al. [58].

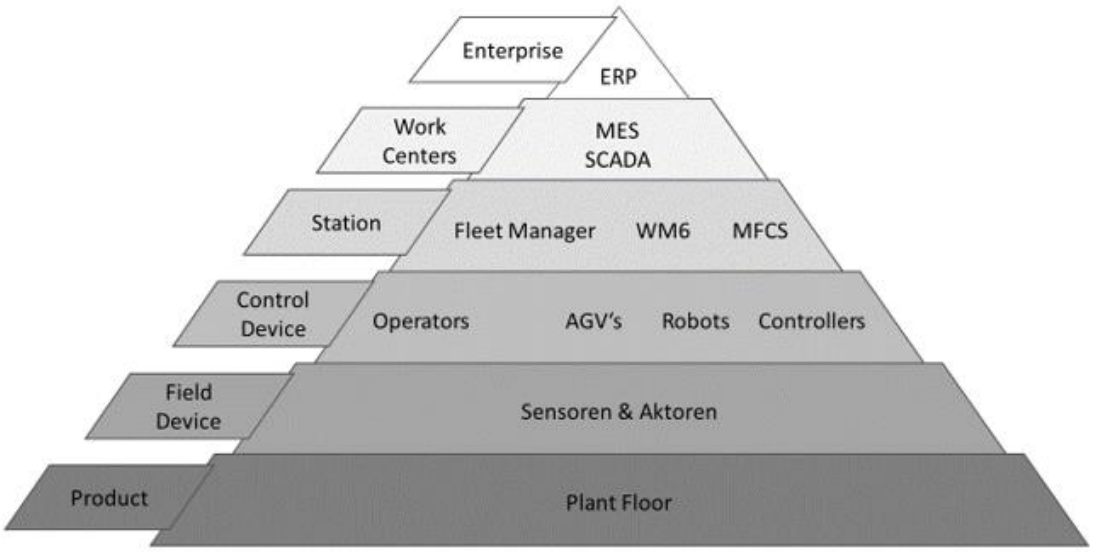

Abbildung 3.2: mögliche Einordnung eines FTS im Ebenen-Modell nach Munz et al. [58]

Die Datenübertragung zu übergeordneten Host-Systemen erfolgt bei Ethernet-basierten Netzwerken vorwiegend über das TCP/IP Protokoll. Als Datenformat wird in diesem Fall weitgehend das Extensible Markup Language (XML) Schema verwendet. Diese Form der Datenübertragung wird jedoch inzwischen als veraltet angesehen. Die Datenübertragung modernerer Anlagen läuft heutzutage größtenteils über Webtechnologien ab. In diesen Fall verfügt der Flottenmanager über ein Application Programming Interface (Web-API) [5]. Dadurch können Daten mittels Representational State Transfer (REST) über HTTP mit der FTS-Leitsteuerung ausgetauscht werden.

Informationen, welche im Zuge einer Transportanforderung durch den Benutzer oder Auftraggeber vom übergeordneten System an die FTS-Leitsteuerung übermittelt werden, sind [5]:

- Transportaufträge mit ID, Quelle, Senke, Priorität und ggf. Ladungsträgeridentifikation
- Änderungen von Transportaufträgen
- Statusanfragen
- Zeitsynchronisation

Im Gegenzug erhält das übergeordnete System von der FTS-Leitsteuerung Rückmeldungen bzgl. der Transportaufträge, Systemstatus, Störungsmeldungen der FTF, usw. [5].

3.3 Schnittstelle zu den Fahrzeugen über die VDA 5050

Viele FTS-Anbieter bieten proprietäre Lösungen, welche zu einer Abhängigkeit vom Anbieter führen. Jedoch kann ein Anbieter bei größeren Systemen nicht alle Anforderungen erfüllen, weshalb eine offene Schnittstelle immer mehr an Bedeutung gewinnt. Eine offene Schnittstelle der Software ermöglicht weiters eine zukünftige Anbindung neuer Fahrzeuge, unabhängig vom Hersteller, an einen vorhandenen Leitstand. Somit wird die Integration von Fahrzeugen an ein bestehendes FTS vereinfacht. Dies ermöglicht einen Parallelbetrieb von FTF unterschiedlicher Hersteller in demselben Arbeitsumfeld unter Verwendung eines Flottenmanagers [53]. Hinsichtlich einer standardisierten Schnittstelle zwischen Fahrzeug und FTS-Leitsteuerung setzt die VDA 5050 (VDMA) erste Standards. Die Vorteile einer solch standardisierten Schnittstelle wären nach der VDA 5050 [53]:

- Reduzierung der Komplexität und Steigerung der „Plug&Play“ Fähigkeit des Systems durch Verwendung einer einheitlichen, übergreifenden Koordinationsebene mit den dazugehörigen Logiken für alle Transportfahrzeuge, Fahrzeugtypen und Hersteller
- Verkürzung der Implementierungszeit durch hohe „Plug&Play“-Fähigkeit. Benötigte Informationen (Karten) werden durch zentrale Services bereitgestellt und sind allgemeingültig.
- Integration von proprietären FTS-Bestandteilen durch vertikale Kommunikation zwischen der proprietären Leitsteuerung und der übergeordneten FTS-Leitsteuerung (siehe Abbildung 3.3).
- Erhöhung der Flexibilität durch u. a. Steigerung der Autonomie der Fahrzeuge, Schnittstellen und Prozessbausteine.

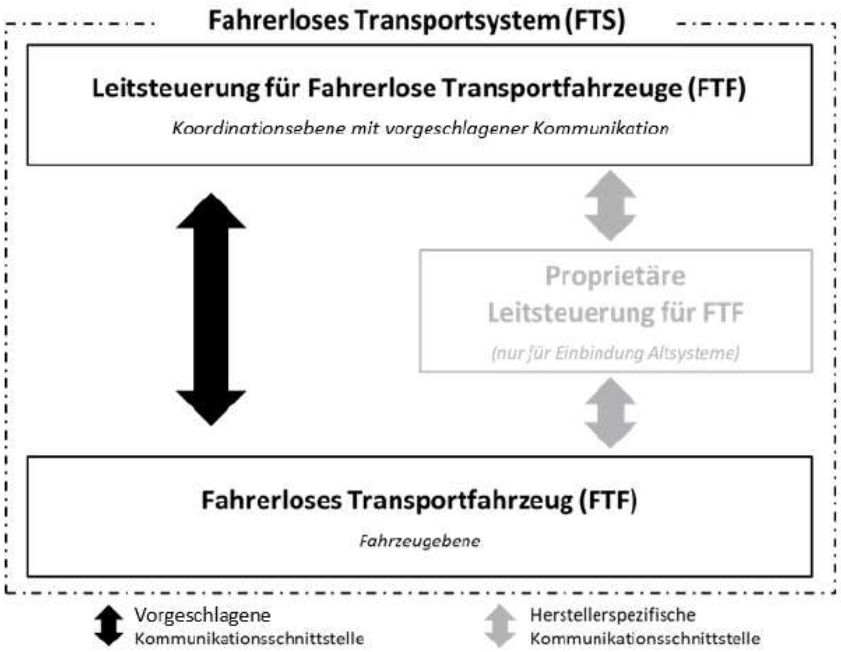

Abbildung 3.3: Einbindung von FTS Bestandsystemen nach der VDA 5050 [53]

Bezüglich der Umsetzung einer solch standardisierter Schnittstelle muss als erster Schritt eine Schnittstelle für die Kommunikation von Auftrags- und Statusinformationen zwischen FTF und Leitsteuerung definiert werden. Hinsichtlich des Informationsflusses für den Betrieb von fahrerlosen Transportfahrzeugen gibt es gemäß der VDA 5050, dargestellt in Abbildung 3.4, mindestens drei Akteure:

- Ein Betreiber, welcher elementare Informationen zur Verfügung stellt, z. B. Kartenmaterial
- Eine Leitsteuerung, die den Betrieb organisiert und verwaltet
- Das FTF, zum Ausführen der Aufträge

In der Konfigurationsphase werden die Leitsteuerung und das Fahrzeug eingerichtet. Die dazu notwendigen Rahmenbedingungen werden durch den Entwickler festgelegt und die benötigten Informationen entweder manuell eingetragen oder durch Import aus anderen Systemen eingefügt. Dies betrifft im Wesentlichen folgende Inhalte [53]:

- Definition von Fahrkursen: Fahrkurse können vom Entwickler manuell oder alternativ über Computer Aided Design (CAD)-Import in die Leitsteuerung implementiert werden
- Konfiguration des Fahrkursnetzes: Stationen, Be- und Entladestationen, Batterieladestationen, etc., werden innerhalb des Fahrkursnetzwerkes definiert.
- Konfiguration des Fahrzeuges: physikalische Eigenschaften des Fahrzeuges, wie etwa die Größe, verfügbare Ladungsträgeraufnahmen, etc., müssen vom Entwickler hinterlegt werden

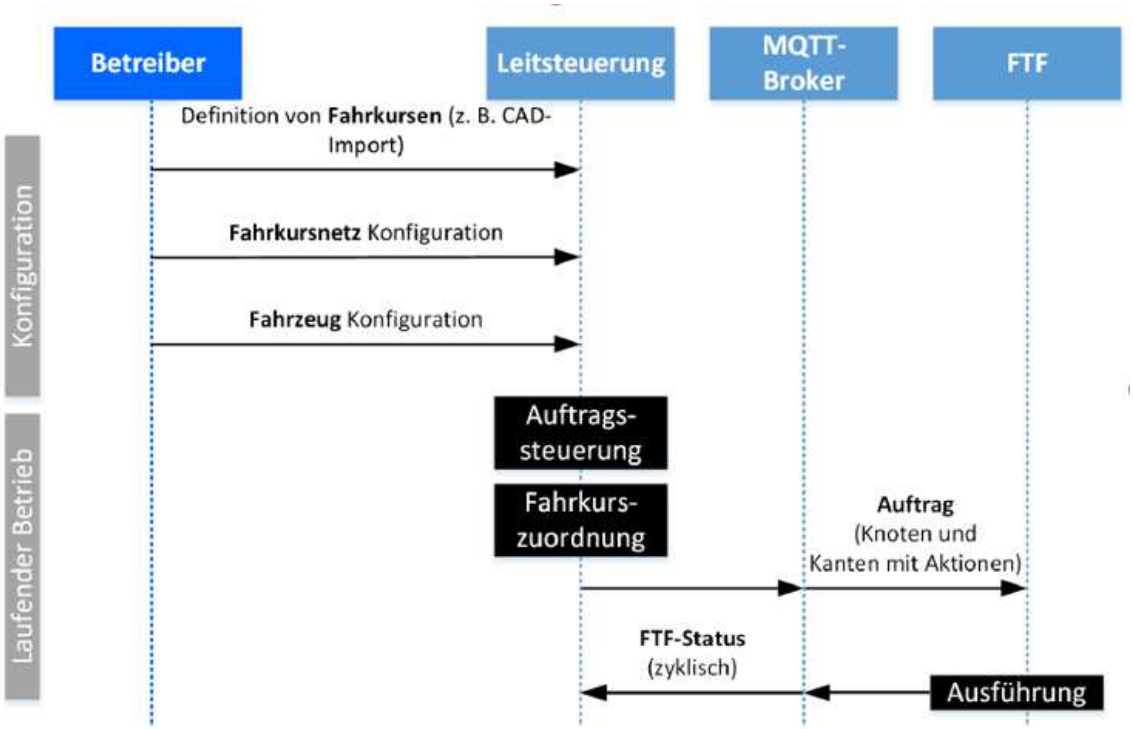

Abbildung 3.4: Struktur des Informationsflusses nach der VDA 5050 [53]

Über einen MQTT Broker können so die erzeugten Transportaufträge an das Fahrzeug übermittelt werden. Anschließend meldet dieser, parallel zur Ausführung des Auftrages, seinen Status über den MQTT Broker an die Leitsteuerung zurück. Somit können die Daten einerseits in Auftragsdaten und anderseits in FTF-Status-Meldungen (zyklisch) unterschieden werden. Mittels des FTF-Status wird die Ausführung der Aufträge bestätigt. Der Status enthält zusätzlich Informationen über die Auftragsübernahme, Positionsmeldung und Batteriewerte. Die Kommunikationsschnittstelle muss in einem weiteren Schritt durch eine zusätzlich definierte Schnittstelle ergänzt werden. Über diese wird ein Kartenaustausch zwischen Leitsteuerung und FTF ermöglicht. Beim Transfer von Karteninformationen ist ein einheitliches Koordinatensystem mit klar definiertem Ursprung für Fahrzeuge mit unterschiedlichen Navigationsarten essenziell [53].

3.4 Flottenmanager-Software

Nachdem die wichtigen Gesichtspunkte wie Funktionen und Schnittstellen einer Leitsteuerung geklärt sind, werden im folgenden Abschnitt Software-Lösungen präsentiert, welche derzeit auf dem Markt verfügbar sind und sich für den Einsatz in der TU-Pilotfabrik eignen.

3.4.1 OpenTCS

OpenTCS (kurz für „open Transportation Control System“) ist eine Open-Source Flottenmanagementsoftware, welche für die Koordination von AGV‘s entwickelt wurde. Dabei steuert OpenTCS die Fahrzeuge unabhängig von ihren spezifischen Merkmalen wie Navigationsprinzip/Spurführungssystem oder Ladesystem. Dabei handelt es sich um kein vollständiges Produkt mit dem AGV‘s "out-of-the-box" gesteuert werden können. Vielmehr ist ein Framework, in dem die grundlegenden Datenstrukturen und Algorithmen, wie Routing, Fahrzeugdisposition und Planung, enthalten sind. Je nach Anforderungen, besteht die Möglichkeit die Algorithmen zu verändern oder projektspezifische Strategien hinzuzufügen. Die Software ist so allgemein wie möglich gehalten. Damit wird die Interaktion mit spurgeführten Fahrzeugen jeden Anbieters ermöglicht. Es ist daher erforderlich, mindestens einen Fahrzeugführer/Adapter zu erstellen und zu integrieren, der zwischen der abstrakten Schnittstelle des openTCS-Kernels und dem Kommunikationsprotokoll, welches das Fahrzeug versteht, übersetzt [59].

Voraussetzung für den Einsatz von OpenTCS

OpenTCS hat keine spezifischen Hardwareanforderungen. Zum Ausführen ist lediglich eine Java Runtime Environment (JRE) erforderlich. Für die Kommunikation mit den Fahrzeugen und möglicherweise anderen Systemen, wie etwa einem Lagerverwaltungssystem, ist eine Art Netzwerkhardware erforderlich. In den meisten Fällen reicht jedoch ein Standard-Ethernet-Controller. Des Weiteren sind Voraussetzung notwendig, um das Fahrzeug mit openTCS verwalten zu können. Zu diesen zählt [59]:

- Eine Kommunikation mit dem Fahrzeug muss möglich sein. Dies bedeutet, dass die Kommunikationsschnittstelle des Fahrzeugs angegeben und zugänglich sein muss.
- Das Fahrzeug muss in der Lage sein, seine aktuelle Position und seinen Status zu melden.
- Das Fahrzeug muss in der Lage sein, Bewegungsbefehle von seiner aktuellen Position zu einer benachbarten Position, in der von openTCS vorgegebenen Fahrstrecke/Umgebung, ausführen zu können.

Aufbau der Software

OpenTCS ist vollständig in der Programmiersprache Java implementiert und somit auf jeder Betriebssystemplattform lauffähig, für die eine Java-Laufzeitumgebung existiert. Die Software kann daher z. B. auf einem Linux-/Unix-System ebenso eingesetzt werden wie auf einem Windows-System. Das gesamte Softwarepaket besteht aus mehreren Komponenten, mit denen ein vollständiges Leitsystem realisierbar ist, dargestellt in Abbildung 3.5.

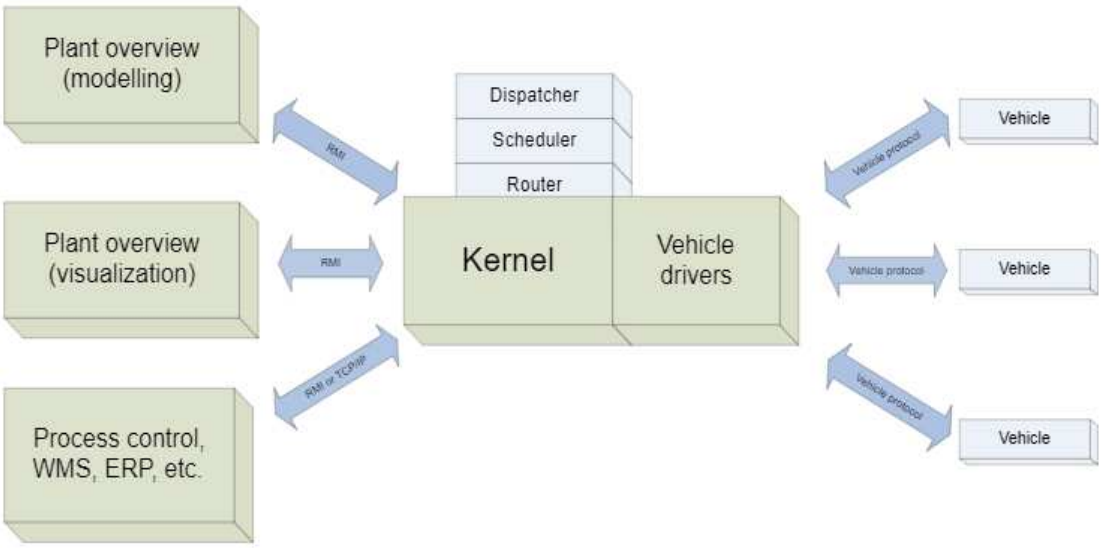

Abbildung 3.5: Komponenten von openTCS [60]

OpenTCS besteht aus den folgenden Komponenten, die als separate Prozesse ausgeführt werden und in einer Client-Server Architektur zusammenarbeiten [60]:

- Kernel: nimmt die zentrale Position im Leitsystem ein. Hier werden der Zustand des Gesamtsystems abgebildet, Transportaufträge an Fahrzeuge vergeben, Fahrtrouten berechnet und Streckenfreigaben erteilt. Außerdem werden Schnittstellen für Bedieneroberflächen, sowie für Hosts, die eine übergeordnete Prozesssteuerung übernehmen, zur Verfügung gestellt.
- Clients
  - PlantOverview, zur Modellierung und Visualisierung des Anlagenmodells
  - Kernel-Kontrollzentrum, zur Steuerung und Überwachung des Kernels, Bereitstellung einer detaillierten Ansicht der Fahrzeuge
  - Beliebige Clients für die Kommunikation mit anderen Systemen zur Prozesssteuerung oder Lagerverwaltung

Außerdem dient die Anlagenübersicht während des Betriebs zur Visualisierung des Anlagenzustandes. Dazu gehört u. a. die Darstellung der gegenwärtigen Fahrzeugpositionen und der zukünftigen Fahrtrouten. Sie kann außerdem zur manuellen Eingabe von Transportaufträgen durch einen Bediener genutzt werden und erlaubt Eingriffe im Störungsfall, wie das Entziehen von Aufträgen.

3.4.2 Kinexon

Das FTS-Leitsystem von Kinexon überwacht und koordiniert die komplette Flotte eines FTS über eine zentrale Plattform. Über diesen ist es möglich, den Fahrkurs zu erstellen und zu ändern. Die Leitsteuerung verfügt über ein dynamisches Routing, somit können im Falle einer blockierten Route Alternativwege eingeschlagen werden.

Ein Vorteil von Kinexon ist die Verwendung von universellen Schnittstellen zwischen FTS-Leitsteuerung und dem Fahrzeug. Dabei wird auf Schnittstellen wie MQTT gesetzt und nach der Protokollbeschreibung der VDA 5050 gearbeitet. Damit soll eine maximale Komptabilität mit verschiedenen Fahrzeugherstellern und sämtlichen gängigen Protokollsprachen geschaffen werden. Bzgl. der Schnittstelle zu übergeordneten Systemen werden Webtechnologien verwendet. Dadurch ist ein Betrieb in größeren Rechenzentren oder einer bestehenden Cloud-Infrastruktur möglich. Weiters verspricht der Hersteller mit einer hohen Anwenderakzeptanz hinsichtlich der Benutzeroberfläche, wodurch sich ein Onboarding ins System recht kurz und einfach stattfinden soll [61].

3.4.3 Incubed IT

Der Flottenmanager von Incubed IT ist eine zentrale Leitsteuerung zur Verbindung der Fahrzeugflotte und der Außenwelt. Zudem werden Dienste zur Optimierung des Verkehrsflusses einer Flotte von autonomen mobilen Robotern angeboten. Zum Verwalten der Kundenprozesse dient die grafische Benutzeroberfläche, über die mit der Flotte interagiert werden kann. Somit wären die zwei wichtigsten Funktionen einer FTS-Leitsteuerung erfüllt. Für den Betrieb kann der Flottenmanagerserver direkt vor Ort oder in einer Cloud-Umgebung installiert werden. Die Installation und weiterführende Konfiguration wird vom Flottenmanager verwaltet. Somit ist keine externe Software für die Gestaltung des Führungswegnetzweges erforderlich. Der Flottenmanagementserver speichert die gesamte Konfiguration der Anlage. Er verteilt bei Bedarf Kartendaten und andere Parameter an alle Fahrzeuge [62].

Weiters übernimmt der Flottenmanager die Überwachung des Verkehrs und die Koordination des Fahrverhaltens. Somit können Streckenbereiche gesperrt bzw. reserviert werden. Definierte Verkehrsregeln werden vom Flottenmanager überwacht. Bezüglich der Kommunikationen zwischen den Fahrzeugen gibt Incubed IT eine zukünftig verfügbare Schnittstelle nach dem Protokoll der VDA 5050. Somit wäre eine Einbindung von FTF von Drittanbietern möglich. Über einen REST-Webservice ist eine Verbindung der IT-Landschaft, wie z. B. ERP oder MES, mit dem Flottenmanager über das Internet möglich. Weitere Möglichkeiten zur Auftragsgenerierung sind direkt über die grafische Oberfläche oder durch weitere regelbasierte Systeme, wie externe Sensoren. Die Zuteilung der Aufgaben erfolgt anhand zahlreicher Eingangsdaten, wie Fahrdistanz oder

Ladezustand der Batterien, somit kann eine Zuteilung von Fahrzeugen zu offenen Aufträgen kalkuliert werden. Dabei werden die Aufträge vom Flottenmanager direkt an das Fahrzeug übermittelt [62].

3.5 Auswahl des Flottenmanagers

Kinexon und Incubed IT bieten einen Flottenmanger mit moderner Benutzeroberfläche an. Weiters ist eine Anbindung an Host-Systeme über REST schnell möglich. Bezüglich der Funktionen gibt es jedoch keine detaillierte Beschreibung. Weiters handelt es sich in beiden Fällen um eine kostenpflichtige Software. Über die Einbindung von Fahrzeugen gibt es keine Informationen darüber, ob es seitens des Unternehmen oder vom Kunden durchgeführt werden muss.

OpenTCS verfügt bereits über viele Standartkomponenten, die ein Flottenmanager benötigt. Neben einem Layout (Führungspfadnetzwerk) benötigt die Software noch einen Fahrzeugtreiber (Kommunikationsschnittstelle), bevor sie praktisch eingesetzt werden kann. Für die Programmiersprache Java gibt es genug Literatur, Bibliotheken und Programmierer, welche eine Implementierung deutlich begünstigen. Als Open-Source-Software können bestimmte Programmteile an das Projekt angepasst und umgeschrieben werden. Somit ist es möglich, die Benutzeroberfläche oder Algorithmen zu verändern bzw. anzupassen.

Aufgrund der oben erwähnten Eigenschaften und der Tatsache, dass keine weiteren Open-Source Flottenmanager vorzufinden waren, fällt die Entscheidung bei der Auswahl der Software auf openTCS. Infolgedessen wird im weiteren Verlauf die Implementierung von openTCS als Flottenmanager in der TU Wien Pilotfabrik beschrieben und durchgeführt.

4 Praxisteil

In diesem Abschnitt wird demonstriert, wie ein Datenaustausch zwischen MES und dem Flottenmanager openTCS stattfinden kann. Weiters wird ein Fahrzeugtreiber (Kommunikationsadapter) in openTCS implementiert, welcher die Kommunikation über die MQTT Schnittstelle zu dem Fahrzeug herstellt.

4.1 Beschreibung des Ausgangszustandes

Wie in der Einleitung erwähnt, steht für die Materialbereitstellung in der Pilotfabrik ein mobiler Roboter (Neobotix MP-400) zur Verfügung. In Abbildung 4.1 werden die Stationen der Produktionsanlagen dargestellt, welche vom AGV zugesteuert werden. Des Weiteren sind Knoten und Kanten in der Abbildung ersichtlich, welche zum Erreichen der Ziele angesteuert werden.

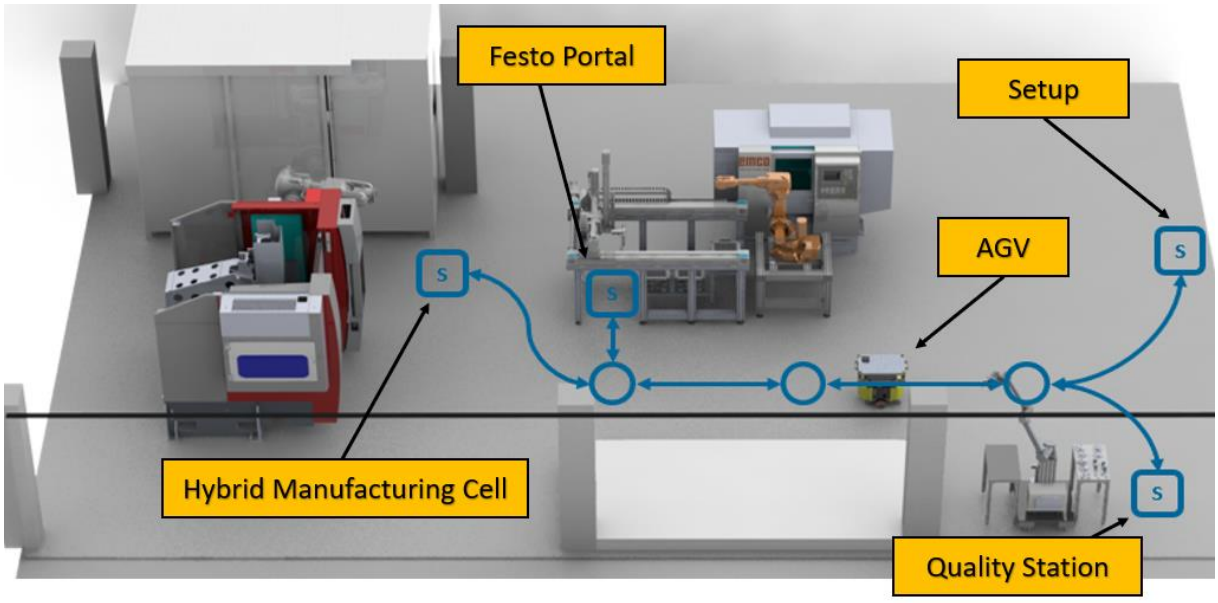

Abbildung 4.1: Materialbereitstellung in der Pilotfabrik durch das FTS, von Thomas Trautner (Abdruck mit Genehmigung)

Der mobile Roboter verfügt über eine freie Navigation, somit fallen physische Leitlinien weg. Führungspfade können somit flexibel, virtuell programmiert werden. Das Fahrzeug besitzt über kein aktives LAM. Infolgedessen wird das Fahrzeug über Roboter und Handhabungsgeräte be- und entladen. Dies geschieht an zwei Stationen, einerseits an der hybriden Fertigungszelle und andererseits am Festo Portal. Das beladene Fahrzeug kann weiters über die FTS-Leitsteuerung an die Station „*Quality Control*“ (QC) befördert werden, wo weitere Arbeitsschritte ausgeführt werden. Neben diesen

drei Stationen existiert noch die Möglichkeit die Aufnahmeplattform des Fahrzeuges über die Station „*Setup*“ zu ändern. Zum Laden des mobilen Roboters steht zusätzlich eine Ladestation zur Verfügung.

Der Roboter verfügt bereits über eine FTS-Leitsteuerung. Jedoch ist diese nicht zum Steuern einer Flotte ausgelegt. Infolgedessen wird ein übergeordneter Flottenmanager (openTCS) implementiert. Die Daten des AGVs werden dabei über den MQTT Broker aus den entsprechenden Topics empfangen. Eine mögliche Implementierung des Flottenmanagers openTCS wird in Abbildung 4.2 beschrieben.

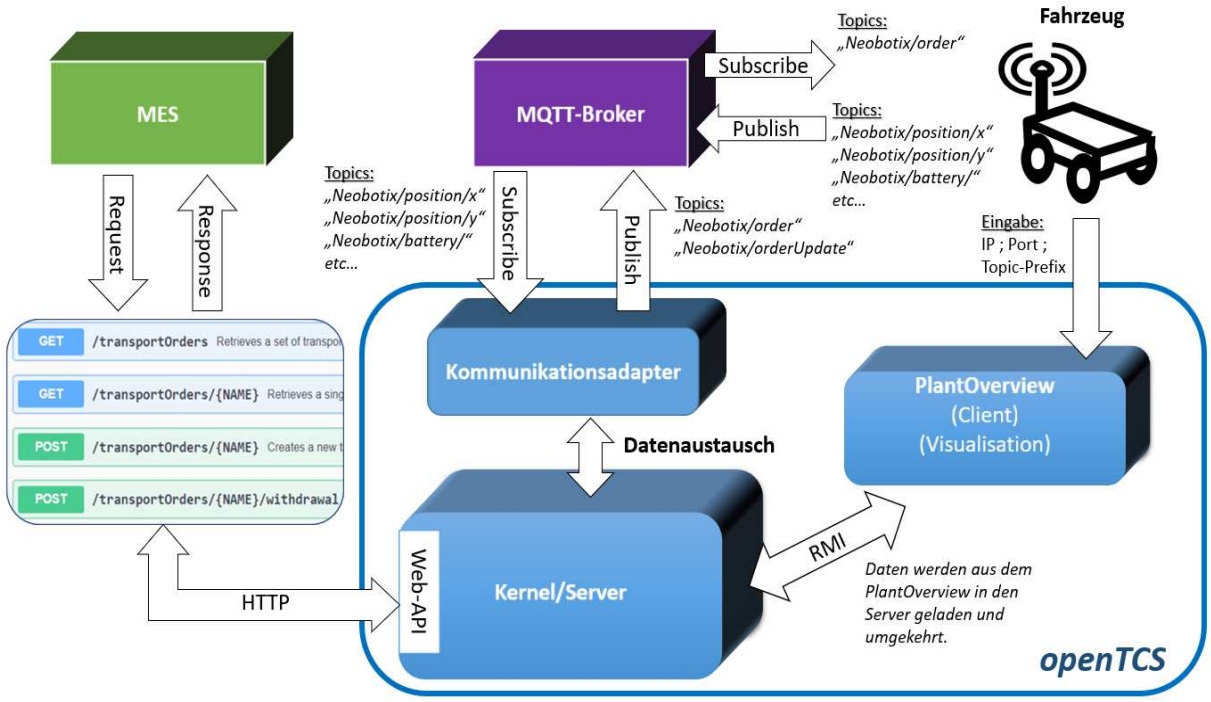

Abbildung 4.2: Integration des Flottenmanagers in die Pilotfabrik

Ein mögliches Szenario für den Prozessablauf würde wie folgt aussehen: Über das ERP System wird ein Auftrag für die Produktion angelegt. Dieser wird nach der Auftragsfreigabe an das MES übergeben. Das MES ist für die Produktion und produktionsnahe Prozesse verantwortlich. Hier werden die notwendigen Arbeitspläne gepflegt und bei Bedarf an die Fertigungszellen übermittelt. Über die Arbeitspläne wird der Materialbedarf bestimmt und bei Nachfrage ein Fahrzeug für den Transport bereitgestellt. Dies erfolgt über die Erstellung eines Transportauftrag, welches über Webservice an die FTS-Leitsteuerung übermittelt wird. Die FTS-Leitsteuerung verwaltet und verarbeitet die Aufträge und sendet die erstellten Fahrbefehle über die entsprechenden Topics an den MQTT Broker. Das vorgesehene Fahrzeug erhält anschließend die Fahrbefehle vom Broker und sendet kontinuierlich seinen Status über ein Topic an die Leitsteuerung zurück. Somit kann der Fortschritt des Auftrages überwacht werden und bei

Bedarf der nächste Teilauftrag gesendet werden. Weiters werden die Fahrzeugdaten im Flottenmanager zum Zweck der Visualisierung der Fahrzeugposition und des Batteriestatus verwendet. Die Abholung des TS bei Fertigstellung der Bearbeitung erfolgt nach demselben Schema. Durch die Fertigmeldung der Bearbeitungszelle wird ein Transportauftrag generiert und versendet. Alternativ könnte über die Dauer des Fertigungsprozesses der Startzeitpunkt des Transportauftrags festgelegt werden. Jedoch wird der Startzeitpunkt eines Transportauftrages durch openTCS nicht unterstützt, und müsste nachträglich programmiert werden. Die Aufträge werden somit direkt nach dem Empfang in openTCS ausgeführt.

4.2 Gestaltung des Führungspfadnetzwerkes

Das Design der Führungspfade ist ein wichtiges Thema bei der Konzeptionierung eines FTS. Es gehört zu den Problemen, welche als eines der allerersten berücksichtigt werden muss. Nach einen kurzen theoretischen Exkurs zur Gestaltung von Führungspfadnetzwerken, wird das Führungspfadnetzwerk für die Pilotfabrik, für den Einsatzfall zweier Fahrzeugen gestaltet.

4.2.1 Aspekte für die Gestaltung der Führungspfade

Bei der Erstellung von Führungspfaden wird prinzipiell nach Gourgand [63] zwischen zwei Kategorein unterschieden:

- Entwurf des Führungsweges und des Standorts der Lastübergabestation auf Grundlage eines bereits vorhandenen Anlagenlayouts
- Entwurf des Führungspfades basierend auf einem vorhandenen Standort der Lastübergabestation und dem Analgenlayout.

Darüber hinaus hängt die Konfiguration des Netzwerkes mit den Kosten für die Pfadkonstruktion, die Platzkosten und die Steuerungskosten zusammen. Ein komplexes Führungspfadsystem, welches ein ausgeklügeltes Steuerungssystem verwendet, erhöht die Kosten erheblich, verringert jedoch die Reisezeit. Daher sollte der Entwickler versuchen die Gesamtlaufzeit des Transportgutes so weit wie möglich zu reduzieren, während das Layout des Führungssystems so einfach wie möglich gehalten werden soll [63]. Guide-Path Systeme können grob, wie in Tabelle 7, nach bestimmten Eigenschaften eingeteilt werden.

Tabelle 7: Eigenschaften von Guide-Path Systemen nach Le-Anh und Koster [64]

Fluss-Topologie	Anzahl der Parallelspuren	Flussrichtung
▪ Konventionell ▪ Single-loop ▪ Tandem	▪ Single lane ▪ Multiple lanes	▪ Unidirectional Flow ▪ Bidirectional Flow

Die Fluss-Topologie beschreibt die Komplexität des Netzwerkes der Führungspfade. Im einfachsten Fall besteht das System aus einer einzigen Schleife. Unter einer Tandemkonfiguration werden mehrere Schleifen, die zusammen gruppiert sind, bezeichnet. Ein konventionelles Layout ist ein kompliziertes Netzwerk mit entsprechenden Pfaden, Überschneidungen, Abkürzungen und Kreuzungen. Ein Pfadsegment in einem Netzwerk kann eine Spur oder einige parallele Spuren enthalten. Weiters wird zwischen Pfadsegmente unterschieden welche nur in eine Richtung (unidirektional) oder in beide Richtungen (bidirektional) durchfahren werden können [64].

Konventionelle Pfadführungssysteme werden in zwei Kategorien eingeteilt, nämlich unidirektionale und bidirektionale Systeme. Ein unidirektionaler Führungspfad ist in der Praxis besonders in Lagern und Verteilzentren weit verbreitet. Das konventionelle bidirektionale Führungspfadsystem ist für den Materialtransport nicht beliebt, obwohl es zu einer höheren Produktivität führen kann als die entsprechende unidirektionale Pfadführung. Grund dafür ist, dass das Steuerungsproblem in einem solchen System sehr kompliziert wird. Derartige Probleme können meist durch Verwendung von zwei unidirektionalen Spuren gelöst werden. Der Nachteil dabei ist, dass deutlich mehr Platz benötigt wird, welcher selten verfügbar ist. Bidirektionale Systeme werden besonders in Systemen verwendet, wo eine Fahrzeugstörung recht selten ist [64].

4.2.2 Erstellen des Führungspfadnetzwerkes

Über den Client „Plant-Overview“ bietet openTCS eine Benutzeroberfläche an, welche es dem Entwickler ermöglicht, ein Anlagenlayout zu erstellen. Diese enthält Tools, womit Knoten, Kanten, Stationen und Fahrzeuge erzeugt und konfiguriert werden können. Das Erstellen des Anlagenlayouts wurde dabei in drei Schritten durchgeführt:

- Erstellen des Führungspfadnetzwerkes
- Hinzufügen und Konfigurieren der Stationen
- Hinzufügen und Konfigurieren von Fahrzeugen

Basis für die Lage der Knoten und Kanten bildete die vorhandene Karte der Pilotfabrik. Anhand dieser Vorlage wurden die Knoten und Stationen erstellt und über Kanten miteinander verbunden. Dabei wurde die Karte leicht verändert, um dem Einsatz zweier Fahrzeuge zu schildern. Wegen des eng bemessenen Bauraumes bot sich für diesen Fall ein konventionelles bidirektionales Führungspfadnetz am besten an.

Beim Anfahren der Ladestation muss beachtet werden, dass die Kontakte für das Laden sich auf der Vorderseite des Fahrzeuges befinden. Somit muss das Fahrzeug von der Ladestation rückwärts wegfahren. Das Rückwärtsfahren von gewissen Strecken wird durch nicht gefüllte Pfeile symbolisiert. Das fertige Layout wird in Abbildung 4.3 dargestellt und besteht aus den fünf bekannten Arbeitsstationen, welche über Knoten und Kanten miteinander verbunden wurden. Weiters wurde eine Park-Position für ein zweites (virtuelles) Fahrzeug erstellt (Point-0019).

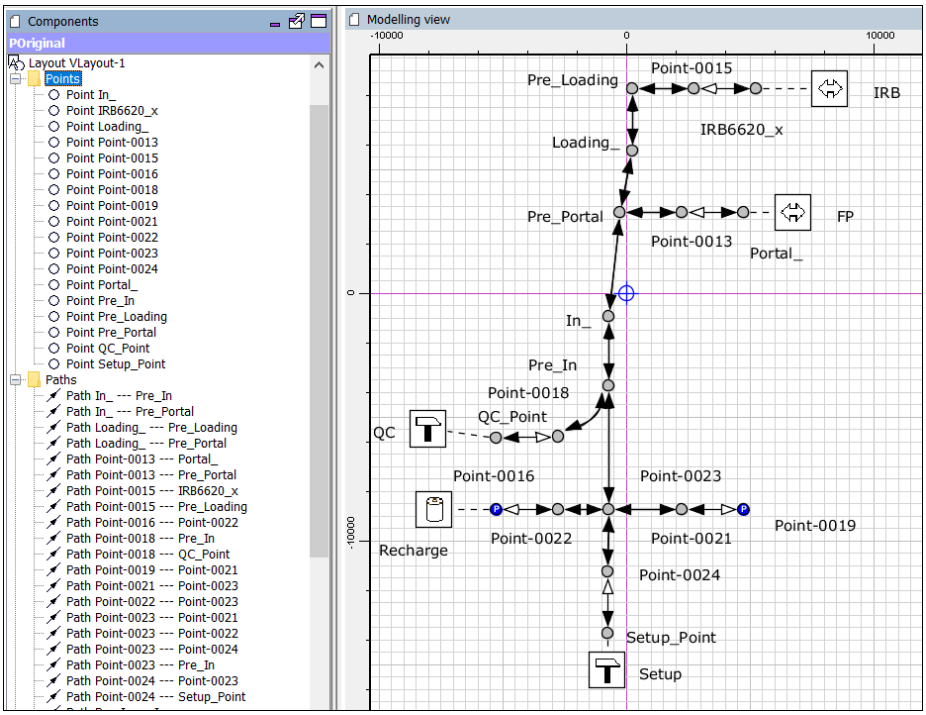

Abbildung 4.3: erstellte Karte für den Betrieb zweier Fahrzeuge für die Pilotfabrik

Nachdem das Führungspfadnetzwerk fertiggestellt ist, werden die Arbeitsstationen konfiguriert, sodass es im späteren Verlauf möglich ist an den jeweiligen Stationen Operationen durchführen zu können. Hierzu muss zuerst ein Location-Typ erstellt werden. Diese sind abstrakte Elemente, die Stationen gruppieren. Ein Location-Typ hat dabei ein relevantes Attribut, welches eine Reihe zulässiger Operationen definiert, die ein Fahrzeug an diesen Orten dieses Typen ausführen darf.

Im weiteren Verlauf wird die Erstellung und Konfiguration des Location-Typ für das Be- und Entladen durchgeführt. Durch Anklicken auf den Location-Type Button wird ein neuer Location-Typ erzeugt und ist unter den Komponenten ersichtlich (siehe Abbildung 4.4). Durch Auswählen dieser öffnet sich das Eigenschaften-Fenster. Hier wird der Name des Location-Typ mit „*Transfer*“ definiert und optional ein Symbol zugewiesen. Durch Anklicken des Textfeldes neben Actions öffnet sich ein Fenster. In diesem Fenster werden die Operationen, welche an der Station ausgeführt werden sollen, eingetragen. In diesem Fall ist es das Be- und Entladen des FTF, welches mit „*Load cargo*“ und „*Unload cargo*“ definiert wird.

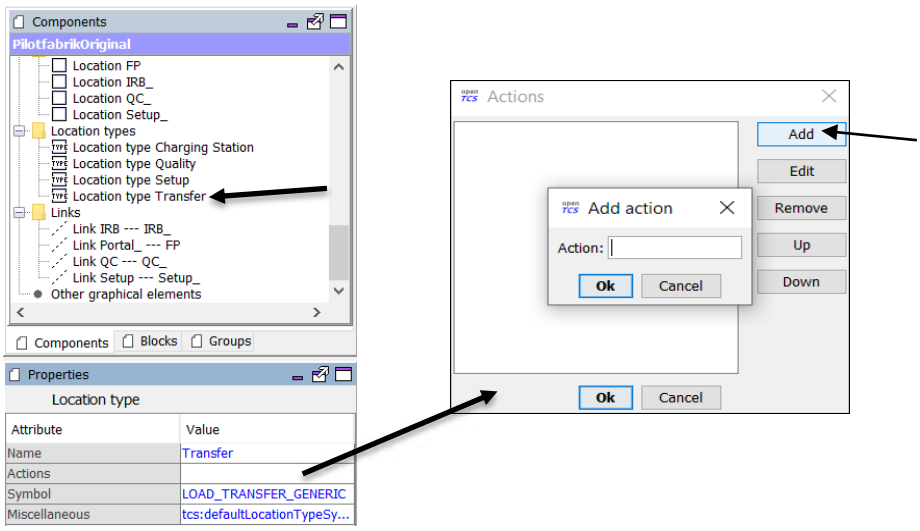

Abbildung 4.4: Konfigurieren der Location-Type

Nachdem die Eingabe bestätigt wurde, sind die Operationen sowohl im Actions-Fenster auch als unter den Eigenschaften ersichtlich. Anschließend muss der erstellte Location-Type den Stationen zugewiesen werden wo eine Be- und Entlade Operation ausgeführt wird. Dies geschieht beim der Hybriden-Fertigungszelle und beim Festo-Portal. Durch Auswählen dieser Stationen im Layout, kann im Eigenschaften-Register mittels des Dropdown-Menüs der erstellte Location-Type den Stationen zugewiesen werden. Die Erstellung und Zuweisung weiterer Location-Types erfolgt simultan nach dem gleichen Schema. Abschließend werden zwei Fahrzeuge erstellt und konfiguriert. In dem Eigenschaften-Fenster können Attribute, wie Name und Länge des Fahrzeuges, Batteriestrategien und zusätzliche Parameter des Fahrzeuges definiert werden. Über die Schaltfläche „*Sonstiges*“ ist es möglich Fahrzeug-Attribute als Schlüssel-Wert Paare zu definieren (siehe Abbildung 4.5). Diese können dann bspw. von Fahrzeugtreiber oder Client-Software gelesen und ausgewertet werden. Sowohl der Schlüssel als auch der Wert können beliebige Zeichenfolgen sein.

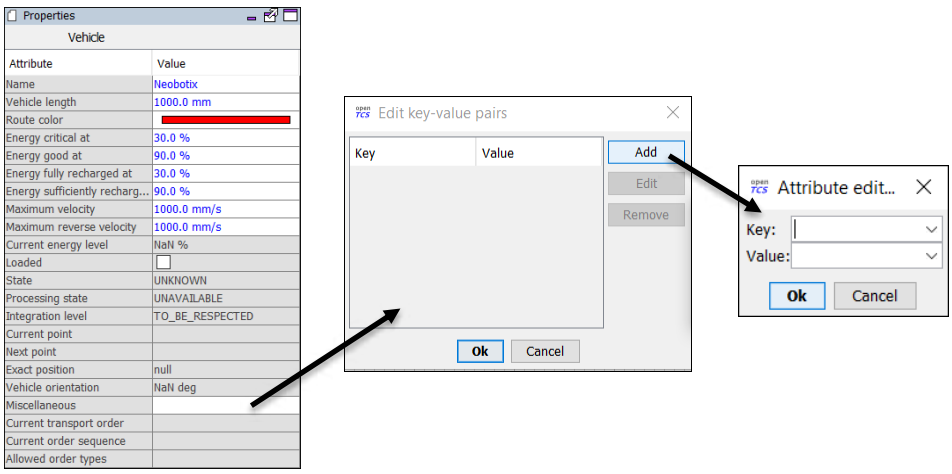

Abbildung 4.5: Erstellen eines Schlüssel-Wert Paares für ein Fahrzeug

Im Fahrzeugtreiber kann dann überprüft werden, ob ein Wert für den Schlüssel vorliegt. Ist dieser vorhanden, kann vom Treiber der Kommunikationskanal zum Fahrzeug automatisch konfiguriert werden. Für den Praxisteil wird ein Schlüssel-Wert Paar für das Be- und Entladen benötigt. Dafür gibt openTCS den Schlüssel bereits vor. Die eingegeben Werte müssen mit dem des Stationstyp äquivalent sein. Somit wird „*Load cargo*“ und „*Unload cargo*“ als Wert festgelegt (siehe Abbildung 4.6).

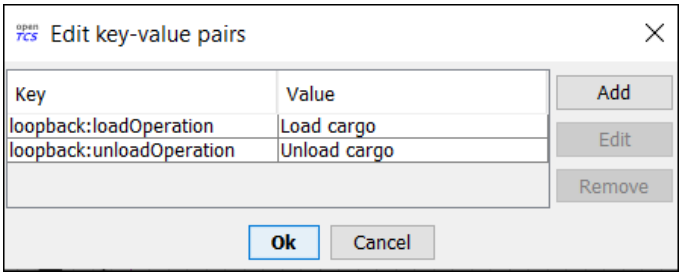

Abbildung 4.6: Erstellen eines Schlüssel-Wert Paar für das Be- und Entladen des Fahrzeuges

Nachdem die Konfiguration abgeschlossen ist, erfolgt eine Simulation über den virtuellen Fahrzeug-Adapter. Dazu müssen zu Beginn der Kernel und das Kernel Control Center (KCC) gestartet werden. Danach wird über den PlantOverview-Client das erstellte Anlagenlayout in den Kernel geladen. Im Anschluss wird der Betriebsmodus des PlantOverviews von „*Modellierungsmodus*“ auf „*Operationsmodus*“ umgestellt. Nachdem dieser Schritt durchgeführt wurde, werden im KCC die zwei erstellten Fahrzeuge angezeigt (siehe Abbildung 4.7). Da vorerst kein Fahrzeugtreiber erstellt wurde, ist derzeit einzig die Auswahl des virtuellen Fahrzeugtreibers möglich. Infolgedessen muss der Startpunkt (Position) der Fahrzeuge angeben werden. Ab diesem Zeitpunkt ist das FTS für eine Simulation bereit.

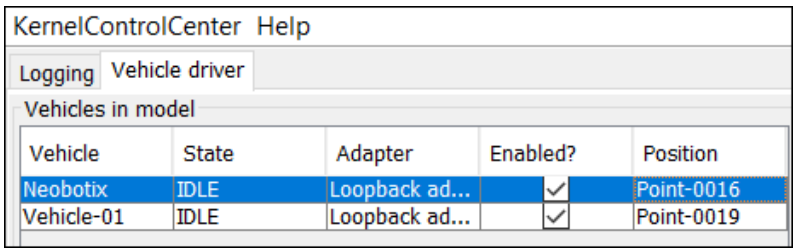

Abbildung 4.7: wählen des Adapters und definieren des Startpunktes der Fahrzeuge

4.2.3 Simulation des Führungspfadnetzwerkes

In der Simulation werden mögliche Szenarios getestet, um Blockaden und Schwachstellen des Anlagenlayouts identifizieren zu können. Zum Testen des Layouts wird der virtuelle Fahrzeugtreiber der Software verwendet, um mögliche Schwachstellen im Layout zu identifizieren. Aufgrund der bidirektionalen Pfadführung ist ein „Deadlock“ für dieses Layout mit mehreren Fahrzeugen unvermeidlich. Coffman [65] definiert ein Deadlock als:

„eine Situation, in der ein oder mehrere gleichzeitige Prozesse in einem System für immer blockiert sind, weil die Ressourcenanforderungen der Prozesse niemals erfüllt werden können“.

Im Falle dieses Layouts wurden zwei Arten von Blockaden festgestellt. Der erste Fall tritt nahezu immer auf, wenn sich auf einem bidirektionalen Pfad zwei Fahrzeuge entgegengesetzt bewegen (siehe Abbildung 4.8).

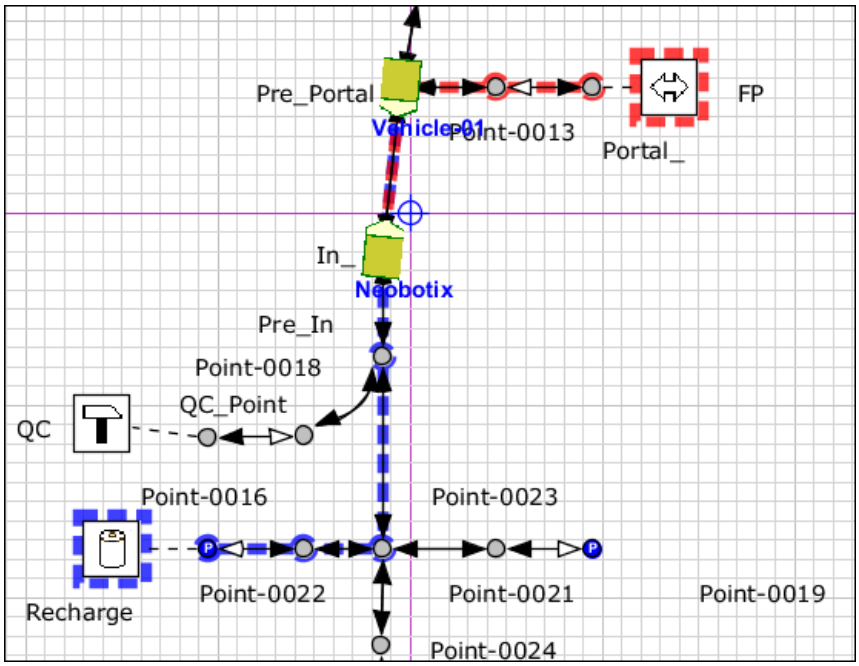

Abbildung 4.8: Entstehung eines Deadlocks durch entgegenfahrende Fahrzeuge

Aus der Abbildung ist zu sehen, dass Fahrzeug Neobotix zur Station „FP“ fährt und Vehicle-01 sich entgegen dieser, in Richtung der Station “Recharge“ bewegt. Das Erstellen einer alternativen Route zum Ausweichen würde in diesem Fall keine Lösung bieten, da openTCS standardmäßig einen statischen Algorithmus zum Berechnen des kürzesten Pfades verwendet. Somit würden die Fahrzeuge bei Vorhandensein einer Ausweichroute dennoch denselben Weg einschlagen und somit das System blockieren.

Um dieses Problem zu beheben wurde ein Blockbereich erstellt. Blockbereiche sind Bereiche, für die besondere Verkehrsregeln gelten. Sie können eingesetzt werden, um Deadlock-Situationen zu verhindern, z. B. an Pfadkreuzungen oder Sackgassen. Dabei hat ein Block zwei relevante Attribute [59]:

- Eine Gruppe von Komponenten (Knoten, Kanten und Stationen) aus denen der Block besteht.
- Einen Typ, welcher die Regeln für die Einfahrt in den Blockbereich wie folgt festlegt:
  - Nur ein einzelnes Fahrzeug
  - Nur dieselbe Richtung

Infolgedessen wurde ein Blockbereich erstellt, der dazu führen soll, dass zwei entgegenfahrende Fahrzeuge auf der Hauptstrecke nicht das System blockieren (siehe Abbildung 4.9).

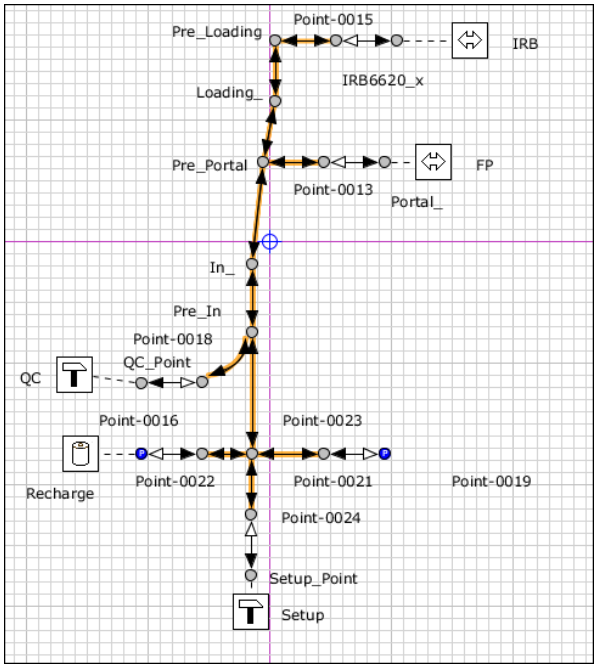

Abbildung 4.9: Erstellter Blockbereich im Anlagenlayout (orange markierte Pfade)

Der Blockbereich wird anhand des Beispiels aus Abbildung 4.10 erläutert. In der Abbildung ist zu erkennen, dass sich die Fahrrouten beider Fahrzeuge überschneiden. Aus dieser Lage heraus würde prinzipiell ohne einen Blockbereich ein Deadlock entstehen. Mithilfe des Blockbereichs wartet das Fahrzeug „Vehicle-01“ bis das Fahrzeug „Neobotix“ den Blockbereich verlässt. Erst dann betritt das Fahrzeug Vehicle-01 den Blockbereich.

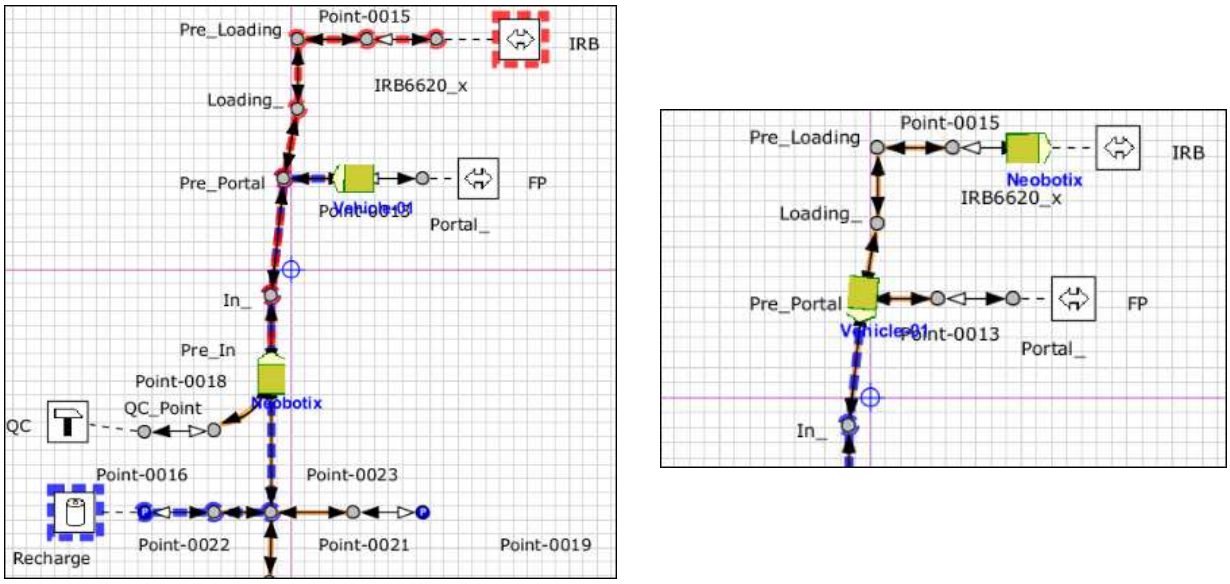

Abbildung 4.10: Vermeidung eines Deadlocks durch den Einsatz eines Blockbereichs

Der zweite Fall der Blockade entsteht durch den Versand von mehreren Fahrzeugen an dieselbe Station. Somit blockiert das einfahrende Fahrzeug die Ausfahrt des anderen Fahrzeuges und umgekehrt (siehe Abbildung 4.11). Diese Art von Deadlock kann prinzipiell an jeder Station auftreten.

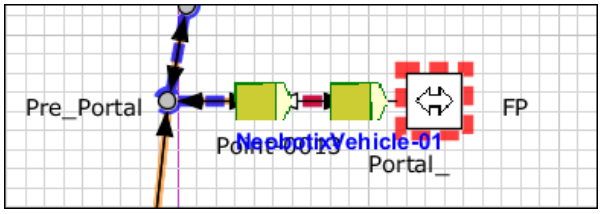

Abbildung 4.11: Entstehung eines Deadlocks durch den Versand von mehreren Fahrzeugen an eine Station

Zur Verbesserung des oben genannten Deadlock-Problems besteht die Möglichkeit der Versetzung der Auftragserteilung. Somit muss das übergeordnete System festlegen, wann die Ressource Station „FP“ zur Verfügung steht. Eine weitere Möglichkeit besteht darin, die Versetzung der Auftragserteilung in openTCS zu implementieren. Dazu muss der Planungs-Algorithmus von openTCS erweitert werden, sodass überprüft werden muss, ob die Station in nächster Zeit belegt wird und somit die Ressource verfügbar ist.

4.3 Schnittstelle zu übergeordneten Systemen

Für eine Kommunikation mit übergeordneten Host-Systemen wie ERP, MES oder Lagerverwaltungssystemen bietet openTCS eine Web-API und eine TCP/IP-Schnittstelle an. Jedoch ist die Datenübertragung mittels TCP/IP veraltet und wird ab der Version 5.0 von openTCS nicht mehr angeboten. Neben diesen Methoden ist auch eine Transportauftragserstellung über die Benutzeroberfläche möglich.

Da für diese Arbeit kein MES zur Verfügung steht, wird für Demonstrationszwecke die Software Node-Red verwendet. Mithilfe dieser Software ist es möglich, Anwendungsfälle im Bereich des IoT mit einem einfachen Baukastensystem umzusetzen. Infolgedessen kann über Node-Red eine Verbindung zur Schnittstelle von openTCS hergestellt werden, um Daten zu übertragen und zu empfangen.

4.3.1 Kommunikation über die TCP/IP Schnittstelle

Für den Host stehen folgende TCP/IP Schnittstellen für die Kommunikation mit dem Kernel zur Verfügung:

- Eine bidirektionale Schnittstelle zur Erstellung von Transportaufträgen
- Eine unidirektionale Schnittstelle zum Empfangen von Statusmeldungen

Zum Erstellen von Transportaufträgen akzeptiert das Kernel Verbindungen zu einem TCP-Port, welche in der Standardeinstellung auf den Port 55555 konfiguriert sind. Über diese Verbindungen sendet der Host ein einzelnes XML-Telegramm, welches den zu erstellenden Transportauftrag beinhaltet. Anschließend muss der Stream entweder vom Host geschlossen werden oder eine Nachricht mit zwei aufeinanderfolgenden Zeilenumbrüchen (d. h."\r\n\r\n") an den Kernel übermittelt werden. Damit wird bestätigt, dass keine weiteren Nachrichten folgen. Im Anschluss verarbeitet der Kernel das Telegramm und erstellt anhand der Daten die Transportaufträge und übermittelt diese an die Fahrzeuge. Optional kann über die bidirektionale Verbindung eine „Quittung“ in Form eines XML-Telegramm von openTCS an den Host gesendet werden. Der Transportauftrag besteht aus einer parametrisierten Abfolge von Bewegungen und Operationen, welche zum Verarbeiten an ein Fahrzeug übermittelt wird. Zum Anlegen eines Transportauftrages können mittels openTCS folgende Attribute definiert werden [66]:

- Eine Reihe von Zielen, welche vom Fahrzeug dementsprechenden angefahren werden. Das Ziel muss dabei aus einer Arbeitsstation und einer Operation bestehen.
- Eine optionale Frist (Deadline) innerhalb welcher der Transportauftrag ausgeführt wird.

- Eine optionale Auswahl der Kategorie des Transportauftrages, z. B.: Wartung, Transport, etc., ist über die Web-API nicht möglich, kann aber bei einer manuellen Erstellung eines Transportauftrages oder über TCP/IP angeführt werden.
- Ein optional vorgesehenes Fahrzeug für den Auftrag. Falls nichts angegeben wird, wird das Fahrzeug automatisch ermittelt und zugewiesen.
- Optional kann noch ein Satz von Abhängigkeiten enthalten sein. Diese Verweisen auf andere Transportaufträge, welche vor diesem Transportauftrag abgeschlossen sein müssen.

Zum Erstellen und Versenden eines Transportauftrages mittels TCP/IP wurde der Flow in Abbildung 4.12 dargestellt, in Node-Red erstellt.

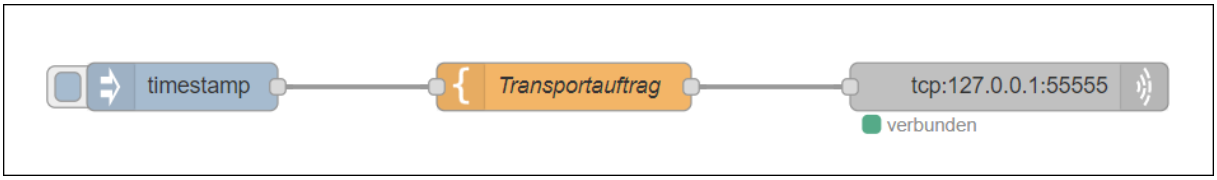

Abbildung 4.12: Erstelltes Flow zur Übertragung von Transportaufträgen über TCP/IP

Das erforderliche XML-Telegramm beschreibt dabei den Transportauftrag und wurde in die Template-Node eingefügt (siehe Abbildung 4.13). Die „*timestamp*“ Node dient dabei als Auslöser. Weiters wurde der Port so konfiguriert, dass die Verbindung automatisch nach dem Senden des Telegrammes schließt. Somit entfällt das Senden von zwei aufeinanderfolgenden Zeilenumbrüchen.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsOrderSet>
    <order xsi:type="transport" deadline="2021-12-10T09:05:39.154+01:00"
        intendedVehicle="Neobotix" id="TransportOrder-10" xmlns:xsi="http
        ://www.w3.org/2001/XMLSchema-instance">
        <destination locationName="IRB" operation="Load cargo"/>
    </order>
</tcsOrderSet>
```

Abbildung 4.13: Beispiel für einen Transportauftrag über TCP/IP mittels XML-Telegramm

Um Statusmeldungen für Transportaufträge empfangen zu können, muss eine Verbindung zu einem TCP-Port des Kernels hergestellt werden, welches in der Standardeinstellung auf Port 44444 konfiguriert ist. Diese Verbindung ist unidirektional, d. h. der

Kernel erwartet keine Nachrichten und verarbeitet keine über diesen Peer empfangenen Daten. Weiters bietet openTCS keine Filterung von Statusmeldungen für den Client an, somit müssen die Telegramme clientseitig gefiltert werden. Wenn sich der Status eines Transportauftrags oder eines Fahrzeuges ändert, wird an jeden verbundenen Client ein XML-Telegramm gesendet, welche den neuen Status des Systems beschreibt. Über den in Abbildung 4.14 dargestellten Flow wurde eine Verbindung mit dem Port des Kernels hergestellt, um Statusmeldungen zu empfangen.

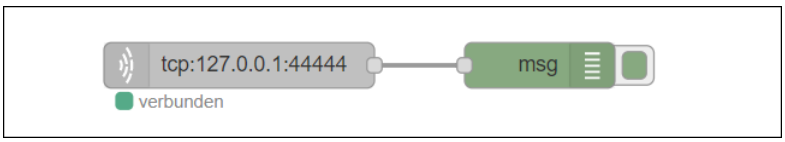

Abbildung 4.14: Erstellter Flow zum Erhalten von Statusmeldungen vom Kernel

Auf jedes erhaltene Telegramm folgt eine Zeichenfolge mit dem Symbol "|", welches das Ende des jeweiligen Telegramms markiert. Die erhaltene Meldung bzgl. des Fortschrittes der Transportaufträge wird in Abbildung 4.15 dargestellt. Über die Information „orderState“ kann der Fortschritt des Transportauftrages gelesen bzw. gefiltert werden.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsStatusMessageSet timeStamp="2020-08-17T11:35:43.521+02:00">
    <statusMessage xsi:type="orderStatusMessage" orderName="TOrder
        -01EFXWCA232T639RGN7BXQX5BR" orderState="FINISHED"
        processingVehicleName="Neobotix" xmlns:xsi="http://www.w3.org/2001
        /XMLSchema-instance">
        <destination locationName="IRB" operation="Load cargo" state
            ="FINISHED"/>
    </statusMessage>
</tcsStatusMessageSet>
```

Abbildung 4.15: Rückmeldung des Transportfortschrittes

Neben dem Fortschritt des Transportauftrages werden auch Fahrzeugdaten bei einer Statusänderung mittels Telegramm an den Peer übermittelt (siehe Abbildung 4.16). Aus diesen Telegrammen können notwendige Informationen bzgl. des Fahrzeuges der MES bereitgestellt werden.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsStatusMessageSet timeStamp="2020-07-08T15:59:36.832+02:00">
    <statusMessage xsi:type="vehicleStatusMessage" position="15"
        processingState="PROCESSING_ORDER" state="IDLE" transportOrderNa
        ="TOrder-01ECQ96DF7JYR1QMFY0WPS27Y6" vehicleName="Neobotix" xmlns
        :xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</tcsStatusMessageSet>
```

Abbildung 4.16: Statusmeldung der Fahrzeuge über die TCP/IP Schnittstelle

Für eine ausführlich Dokumentation des XML-Schemas für die TCP/IP-basierte Hostschnittstelle wird auf die Distribution des Herstellers hingewiesen[1].

4.3.2 Kommunikation über Webservice

Über die Web-API besteht für den Host die Möglichkeit folgende Informationen abzurufen bzw. zu übermitteln:

- Abrufen von Fahrzeugdaten
- Ändern des Integrationslevels eines Fahrzeuges
- Erstellen eines Transportauftrages
- Zurückziehen eines Transportauftrages
- Abrufen von Transportaufträgen
- Status-Abfragen

Die dafür benötigten URLs und Informationen wurden aus der WEB-API Dokumentation des Herstellers entnommen. Für eine detaillierte Beschreibung wird auf diese Distribution hingewiesen[2].

Der verwendete Port für die HTTP-Anforderungen ist konfigurationsabhängig und standardmäßig auf 55200 eingestellt. Das Abrufen von Daten erfolgt über die HTTP-GET Methode. Durch den oberen Flow, dargestellt in der Abbildung 4.17, werden die Daten aller Fahrzeuge aufgerufen. Für diesen Aufruf wird keine Eingabe benötigt, somit wurde die URL direkt in den HTTP-Request eingetragen.

[1] https://www.opentcs.org/docs/4.18/index.html
[2] https://www.opentcs.org/docs/4.18/developer/web-api/index.html#/

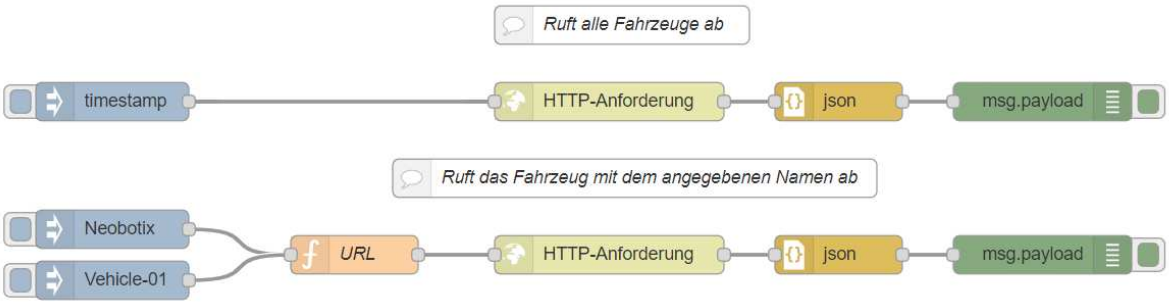

Abbildung 4.17: Flow zum Abrufen von Fahrzeugdaten; oben: Aufrufen aller Fahrzeuge, unten: Aufrufen eines bestimmten Fahrzeuges

Das Aufrufen eines bestimmten Fahrzeuges ist ebenfalls in der oberen Abbildung (unterer Flow) dargestellt. Hierbei wird die Bezeichnung des Fahrzeuges als String eingegeben. Diese Eingabe wird über einen Function-Node an die URL-Adresse angehängt (siehe Abbildung 4.18).

```
msg.url="http://localhost:55200/v1/vehicles/" + msg.payload
```

Abbildung 4.18: URL zum Abrufen von Informationen eines bestimmten Fahrzeuges

Die Antwort von openTCS ist eine JSON-Nachricht (siehe Abbildung 4.19). Aus dieser JSON-Nachricht können Informationen bzgl. des Fahrzeuges abgelesen bzw. relevante Daten herausgefiltert werden.


```
{
  "name": "Neobotix",
  "properties": {
    "tcs:preferredAdapterClass": "org.opentcs.virtualvehicle
       .LoopbackCommunicationAdapterFactory",
    "loopback:unloadOperation": "Unload cargo",
    "loopback:loadOperation": "Load cargo"
  },
  "length": 1000,
  "energyLevelGood": 90,
  "energyLevelCritical": 30,
  "energyLevel": 100,
  "integrationLevel": "TO_BE_UTILIZED",
  "procState": "IDLE",
  "transportOrder": null,
  "currentPosition": "15",
  "state": "IDLE"
}
```


Abbildung 4.19: JSON-Antwort bzgl. der Fahrzeuginformationen

Neben dem Aufrufen von Fahrzeugdaten anhand ihrer Bezeichnung, besteht die Möglichkeit Fahrzeuge, abhängig von ihrem aktuellen Prozessstatus, abzurufen. Dabei definiert openTCS vier unterschiedliche Prozesszustände, die in Abbildung 4.20 ersichtlich sind.

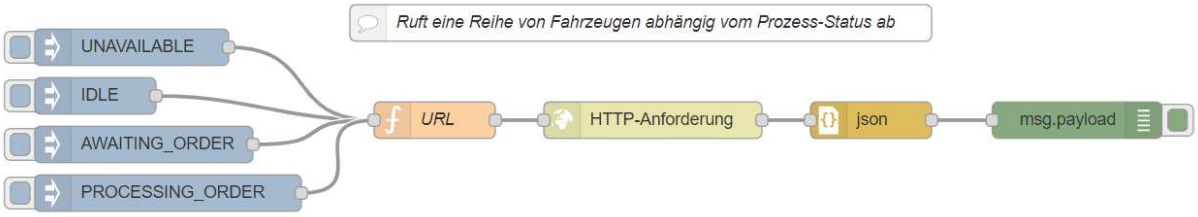

Abbildung 4.20: Abrufen von Fahrzeugen anhand ihres Prozessstatus

Der erstellte Flow läuft nach demselben Schema wie das vorherige Beispiel ab. Die dafür notwendige URL ist in Abbildung 4.21 verzeichnet. Die erhaltene JSON-Nachricht ist in ihrer Form mit dem vorherigen Beispiel äquivalent.

```
msg.url="http://localhost:55200/v1/vehicles?procState=" + msg.payload
```

Abbildung 4.21: URL zum Abrufen von Fahrzeugen anhand des Prozess-Status

Als nächstes wird demonstriert, wie das Integrationslevel der Fahrzeuge verändert wird. OpenTCS legt vier Integrationslevel fest, welche wie folgt beschrieben werden [59]:

- ignored: Das Fahrzeug wird vollständig ignoriert. Somit wird es in der Anlagenübersicht nicht angezeigt und steht für die Annahme von Transportaufträgen nicht zur Verfügung.
- noticed: Die Position des Fahrzeuges wird wahrgenommen und in der Anlagenübersicht angezeigt. Jedoch werden dem Fahrzeugen keine Ressourcen zugeordnet, womit Transportaufträge nicht angenommen werden.
- respected: Die Mittel für die gemeldete Position des Fahrzeugs wird zugeordnet. Das Fahrzeug steht für Transportaufträge nicht zur Verfügung.
- utilized: Das Fahrzeug ist voll einsatzbereit und steht für die Annahme von Transportaufträgen zur Verfügung.

Über die HTTP-PUT Methode kann das Integrationslevel eines Fahrzeuges verändert werden. Dazu wurde ein Flow, dargestellt in Abbildung 4.22, erstellt.

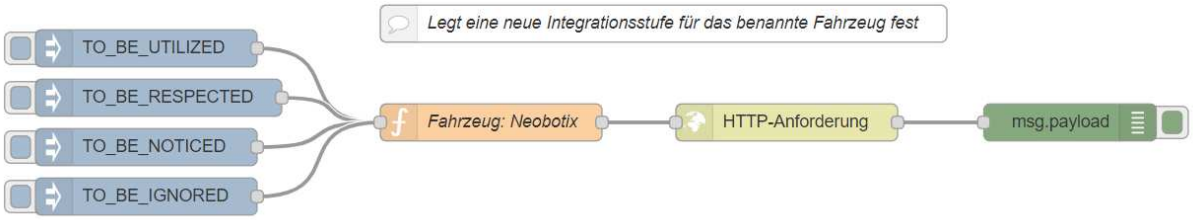

Abbildung 4.22: Flow zum Ändern des Integrationslevels der Fahrzeuge

Die dazu verwendete URL wird in Abbildung 4.23 dargestellt.

```
msg.url="http://localhost:55200/v1/vehicles/Neobotix/integrationLevel
  ?newValue="+msg.payload
```

Abbildung 4.23: URL zum Ändern des Integrationslevel der Fahrzeuge

Durch Ausführen des Flows: „TO_BE_UTILIZED“ wird das Fahrzeug voll einsatzfähig und ist somit im Stande, Transportaufträge anzunehmen (siehe Abbildung 4.24).

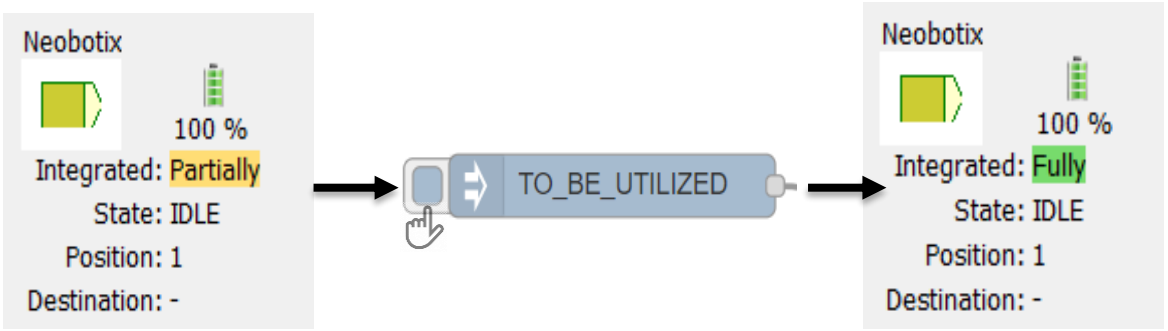

Abbildung 4.24: Ändern des Integrationslevel eines Fahrzeuges

Um von einen bestimmten Fahrzeug den aktuell verarbeiteten Transportauftrag zu widerrufen wird die HTTP-POST Methode aus der Abbildung 4.25 verwendet.

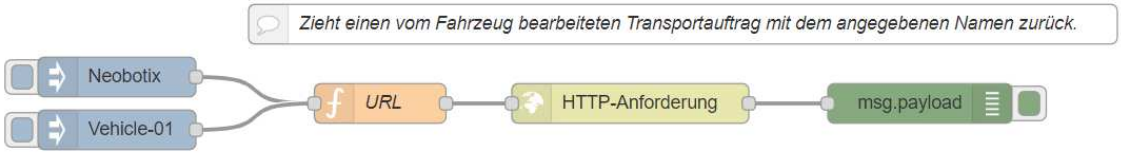

Abbildung 4.25: Flow zum Widerrufen des Transportauftrages eines Fahrzeuges

Beim Widerrufen des Transportauftrages stehen zwei Optionen zur Verfügung. Zum einen kann das Fahrzeug sofort angehalten werden („*withdrawal?immediate=false*“), welches jedoch nicht empfohlen wird. Zum anderen besteht die Möglichkeit das Fahrzeug ausrollen zulassen, durch das Setzten von „*withdrawal?immediate=false*“ (siehe Abbildung 4.26).

```
msg.url="http://localhost:55200/v1/vehicles/"+msg.payload+"/withdrawal
  ?immediate=false&disableVehicle=false"
```

Abbildung 4.26: URL zum Widerrufen eines aktuell verarbeiteten Tranportauftrag von einem Fahrzeug

Die Option „*disableVehicle=false*“ kann dabei den Integrationslevel des Fahrzeuges ändern, womit der Auftrag automatisch abgebrochen wird. Jedoch wird diese Option von OpenTCS als veraltet angegeben und somit auf „false“ gesetzt.

Das Handling mit Transportaufträgen läuft nach dem gleichen Schema ab, wie der Umgang mit Fahrzeugdaten. Einzig das Erstellen von Transportaufträgen erfordert etwas mehr Beachtung. Das Erzeugen eines Transportauftrages wird über die HTTP-POST Methode realisiert. In Abbildung 4.27 wird der erstellte Flow zum Anlegen eines Transportauftrags dargestellt.

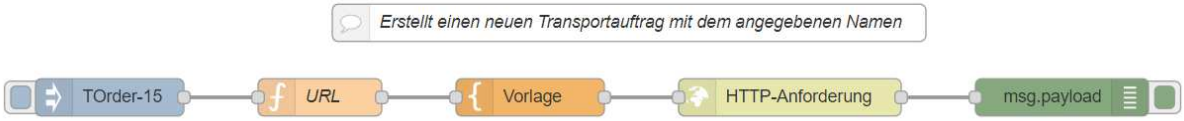

Abbildung 4.27: Flow zum Erstellen eines Transportauftrages

Über den Input wird die Transport-ID des Transportauftrages definiert, welche einmalig auftreten darf. Über den Function-Node wird die URL eingetragen (siehe Abbildung 4.28).

```
msg.url="http://localhost:55200/v1/transportOrders/"+msg.payload
```

Abbildung 4.28: URL zum Anlegen eines Transportauftrages

In der Template-Node wird anschließend der Transportauftrag im JSON-Format eingetragen. Für diese Demonstration wurde ein Transportauftrag mit der ID: „TOrder-15“ erzeugt, welcher ein Fahrzeug zur Station „IRB“ zum Beladen eines TS sendet. Ein auszuführendes Fahrzeug wird dabei nicht angegeben und somit vom Dispatcher ausgewählt. Die dafür geschriebene JSON-Datei ist in Abbildung 4.29 ersichtlich. Die darin enthaltene Deadline wurde frei gewählt.


```
{
  "deadline": "2020-08-17T06:42:40.396Z",
  "destinations": [
    {
      "locationName": "IRB",
      "operation": "Load cargo",
      "properties": [{}]
    }
  ],
  "properties": [{}],
  "dependencies": []
}
```


Abbildung 4.29: erstellter Transportauftrag in JSON

Das Aufrufen eines erstellten Transportauftrag wird, wie bereits erwähnt, über die HTTP-GET Methode verwendet. Einzig die ID des Transportauftrages muss in der URL angegeben werden. Als Rückmeldung sendet openTCS eine JSON-Nachricht, in der alle relevanten Daten zum Transportauftrag enthalten sind (siehe Abbildung 4.30). Anhand dieser JSON-Nachricht kann der Fortschritt des Transportauftrages erfasst werden. Diese Daten können im weiteren Schritt gefiltert und der MES bereitgestellt werden.

```
{
  "name" : "TOrder-15",
  "type" : "-",
  "state" : "FINISHED",
  "intendedVehicle" : null,
  "processingVehicle" : "Neobotix",
  "destinations" : [ {
    "locationName" : "IRB",
    "operation" : "Load cargo",
    "state" : "FINISHED",
    "properties" : [ {
      "key" : "",
      "value" : ""
    } ]
  } ],
  "category" : "-"
}
```


Abbildung 4.30: Antwort-Nachricht als JSON von openTCS bzgl. eines Transportauftrages

Zuletzt ist es noch möglich Statusaktualisierungen abzurufen. Die dafür vorgesehene URL ist in Abbildung 4.31 dargestellt. Über „*minSequenceNo*“ kann die minimale Sequenzanzahl der abzurufenden Ereignisse eingestellt werden. Dies kann verwendet werden, um bereits abgerufene Ereignisse herauszufiltern. Die „*maxSequenceNo*“ wird verwendet, um die Anzahl der abgerufenen Ereignisse zu begrenzen. Zusätzlich wird mit „timeout“ die Zeit in Millisekunden definiert, um auf das Eintreffen von Ereignissen zu warten.

```
msg.url="http://localhost:55200/v1/events?minSequenceNo=0&maxSequenceNo=9223372036854775807&timeout=1000"
```

Abbildung 4.31: URL zum Abrufen des System-Status

4.4 Erstellen einer MQTT Schnittstelle zum Fahrzeug

Da openTCS auf der Java Plattform basiert, erfolgt die Programmierung des Fahrzeugtreibers (Kommunikationsadapter) ebenfalls über Java. Somit müssen zu Beginn Java Development Kit (JDK) und JRE installiert werden. Des Weiteren wird eine Integrated Development Environment (IDE) benötigt. Neben JDK, JRE und einer IDE wird noch als Build-Management Tool Gradle verwendet.

OpenTCS bietet eine Möglichkeit ein Integrationsprojekt einfach zu generieren. Dazu wird zuerst die openTCS-Example-5.0.0-src.zip Datei von der Homepage heruntergeladen und entpackt. Danach wird über die Windows-Eingabeaufforderung das Stammverzeichnis des Projektes gewählt und über den Befehl: *„gradle cloneProject“*, das Projekt gebildet. Das Integrationsprojekt befindet sich nach erfolgreichem Ablauf im Projektordner unter dem „*build/“* Verzeichnis. Da die im Projekt enthaltenen Klassen generische Namen besitzen, ist es ratsam den Namen anzupassen, indem der o. g. Befehl mit einigen Eigenschaften versehen wird. Als Beispiel für diese Arbeit wurde der Integrationsname mit „*Pilotfabrik*“ und der classPrefix mit „*Neobotix*“ betitelt. Somit lautet der vollständige Befehl zum Erstellen des Integrationsprojektes folgendermaßen:

„gradlew -PintegrationName=Pilotfabrik -PclassPrefix=Neobotix cloneProject“

Das erstellte Projekt kann nach erfolgreichem Build über die IDE geöffnet und bearbeitet werden.

Damit openTCS mit dem Fahrzeug kommunizieren kann muss ein Fahrzeugtreiber (Kommunikationsadapter) programmiert werden. Dies erfolgt in drei Schritten:

- Erstellen einer MQTT Schnittstelle zum Abonnieren der Topics aus dem Broker und zuweisen der gefilterten Daten an openTCS
- Funktionserweiterung des Adapters
- Umwandeln der Fahrbefehle aus openTCS in ein JSON-Format nach Vorlage der VDA 5050 und anschließendes Publishen an den Broker

Anmerkung: Im weiteren Verlauf werden nur einzelne Abschnitte des Codes beschrieben. Der vollständig geschriebene Programmcode befindet sich im Anhang. Weiters sei angemerkt, dass hauptsächlich in der Klasse „*NeobotixCommAdapter*“ programmiert wurde und zusätzlich für den Datenempfang und -versand notwenige Klasse erstellt wurden. Die weiteren Dateien im Integrationsprojekt bleiben unverändert.

4.4.1 Einlesen der Daten aus dem MQTT Broker

Um die fahrzeugrelevanten Daten aus dem MQTT Broker abrufen zu können, muss eine MQTT Schnittstelle programmiert werden, in der ein MQTT Client die entsprechenden Topics des Brokers abonniert. Für diesen Zweck wurde die Eclipse Paho Bibliothek importiert und verwendet. Über diese MQTT Schnittstelle können die benötigten. Mit den erstellten Client werden die Topics des Brokers abonniert und erhalten im Gegenzug folgende benötigte Daten des Fahrzeuges:

- X-Position
- Y-Position
- Orientierungswinkel Theta
- Batteriestatus
- Beladungsstatus

Die gesendeten Positionsdaten des Fahrzeuges beziehen sich auf den Ursprung der Karte, welche im Fahrzeug gespeichert ist. Dieser muss, mit der in openTCS erstellten Karte, übereinstimmen. Um eine Verbindung mit dem MQTT Broker aufbauen zu können, wird dessen IP-Adresse und Portnummer benötigt. Diese müssen vom Entwickler über die Fahrzeugeigenschaften im PlantOverview eingegeben werden (siehe Abbildung 4.32).

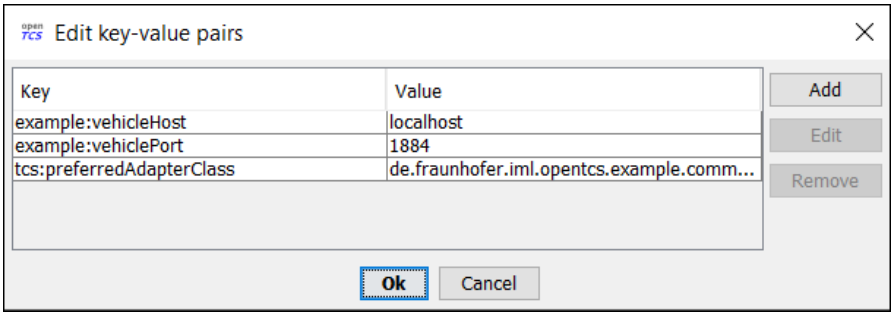

Abbildung 4.32: Eingabe von IP-Adresse und Portnummer über die Fahrzeugeigenschaften

Die Eingabe der IP-Adresse und des Ports über die Fahrzeugeigenschaften ist zwingend erforderlich, da openTCS dies in der Klasse „*ComponentsFactory*“ vorschreibt. Beim Fehlen dieser Angabe wird der programmierte Adapter nicht erkannt und automatisch der Loopback-Adapter geladen. IP-Adresse und Port können im weiteren Verlauf zusätzlich über die Benutzeroberfläche des KCC verändert werden (siehe Abbildung 4.33).

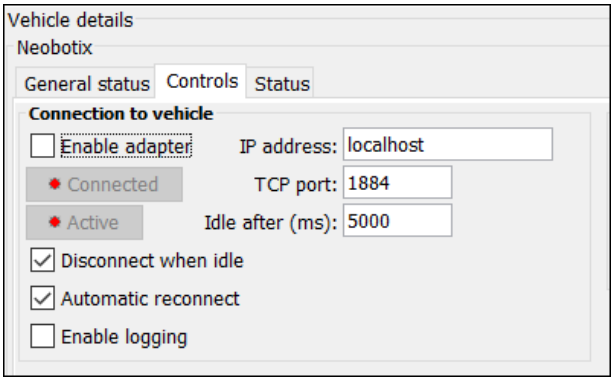

Abbildung 4.33: Eingabe der IP-Adresse und Portnummer über das KCC

Der geschriebene Code zum Abonnieren der Topics ist in Abbildung 4.34 ersichtlich. Darin wurden Triplets (Tupel welche drei Elemente erfasst) definiert, welche die unterschiedlichen Topics, IDs und Callbacks beinhalten.

```
@Override
protected synchronized void connectVehicle() {
  String brokerIp = getProcessModel().getVehicleHost();
  int brokerPort =  getProcessModel().getVehiclePort();
  clientList = new ArrayList<>();
  callbackData = new Triplet[]{
          new Triplet<String,String,MqttCallback>(topicPrefix +
                  "/position/x", "PositionX Client",
                  new CallbackPositionX(this)),
          new Triplet<String,String,MqttCallback>(topicPrefix +
                  "/position/y", "PositionY Client",
                  new CallbackPositionY(this)),
          new Triplet<String,String,MqttCallback>(topicPrefix +
                   "/position/a", "Orientation Angle Client",
                  new CallbackOrientationAngle(this)),
          new Triplet<String,String,MqttCallback>(topicPrefix +
                   "/battery/voltage_percentage", "Battery Client",
                  new CallbackBattery(this)),
          new Triplet<String,String,MqttCallback>(topicPrefix +
                   "/controller/loading", "Load State Client",
                  new CallbackLoadState(this)),
  };
  try {
      for(Triplet<String,String,MqttCallback> t : callbackData){
        MqttClient client = new MqttClient("tcp://"+ brokerIp
                + ":" + brokerPort,t.getValue1());
        client.setCallback(t.getValue2());
        client.connect();
        client.subscribe(t.getValue0(),2);
        clientList.add(client);
      }
```

Abbildung 4.34: Code zum Abonnieren der Topics

Damit zukünftig neue Fahrzeuge mit geringen Aufwand eingebunden werden können, wird ein *Topic-prefix* eingeführt. Voraussetzung dafür ist, dass die nachfolgenden Topic-Leveln äquivalent ist. Somit können die Topics zum Abonnieren unterschiedlicher Fahrzeuge nach Folgenden Schema aufgebaut werden:

- **TopicPrefix/lower topiclevel**
  - Bsp.: Neobotix/position/x
    → Topic-Prefix: Neobotix
  - Bsp.: Pilotfabrik/Neobotix/Fahrzeug2/position/y
    → Topic-Prefix: Pilotfabrik/Neobotix/Fahrzeug/2
  - Bsp.: Omikron/Fahrzeug2/battery
    → Topic-Prefix: Omikron/Fahrzeug2

Die Eingabe des *Topic-Prefix* erfolgt wie die Portnummer und IP-Adresse über die Fahrzeugeigenschaften im PlantOverview. Über Setter und die Methode „*getProcess-Model()*“ können im nächsten Schritt die aus dem PlantOverview ausgelesenen Daten den Fahrzeugtreiber zugewiesen werden (siehe nächstes Kapitel). Um die erhaltenen Payloads vom Broker ausgeben zu können, müssen „*Callbacks*“ erstellt werden. Die Namen der Klassen wurden dabei funktional gewählt, sprich für den Batteriestatus wurde die Klasse „*CallbackBattery*“ erstellt. Die vom Broker gesendete JSON-Nachricht, besteht wie in Abbildung 4.35 dargestellt, aus einem Header und dem Wert, hier Batteriezustand des Fahrzeuges.

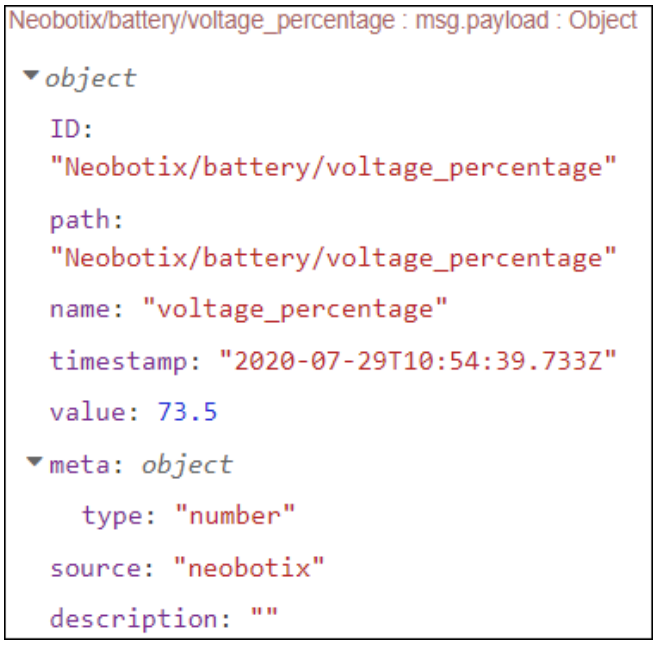

Abbildung 4.35: gesendete JSON-Nachricht vom Broker

Da es sich bei der Payload um ein JSON-Object handelt, muss dieses als erster Schritt in ein Java-Object umgewandelt werden. Für diese Umsetzung wurde die Gson Libary importiert und verwendet. Gson ist eine Java-Bibliothek von Google, mit der Java-Objekte in JSON-Darstellung konvertiert werden können. Um die Daten des Payload Java Klassen zuweisen zu können, wurden dazu die jeweiligen Java Klassen im Ordner „*models*“ erstellt (siehe Anhang). Mit der Funktion „*getProcessModel()*“ können die gefilterten Daten an openTCS übergeben werden. Im Anschluss kann die Software gestartet werden, um die Schnittstelle zu testen. Ist das Anlagenlayout im Operationsmodus und der Adapter im KCC aktiviert, so erscheint das Fahrzeug auf der Karte gemäß den Daten, welche vom Broker empfangen werden (siehe Abbildung 4.36).

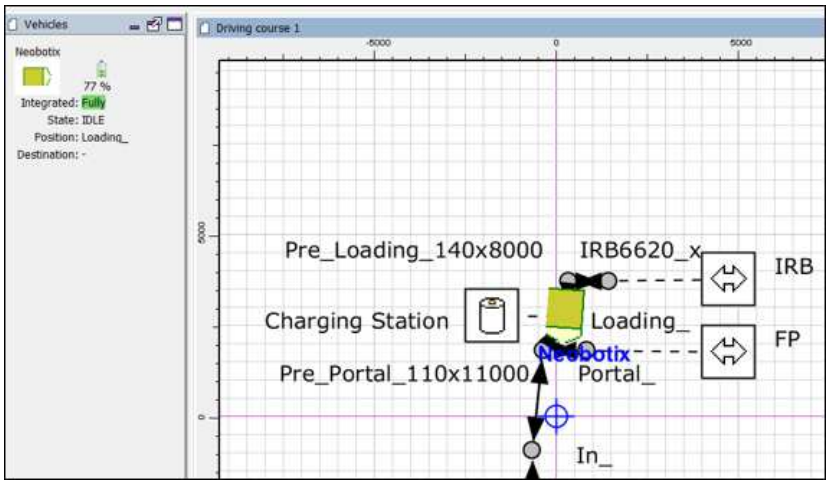

Abbildung 4.36: Anzeige der Fahrzeugposition, Orientierung und Batteriestatus nachdem Einlesen und Zuweisen der Daten aus dem Broker

4.4.2 Funktionserweiterung des Fahrzeugtreibers

Im nächsten Schritt müssen gewisse Funktionen im „*NeobotixCommAdapter*“ erweitert bzw. neu hinzugefügt werden, welche für eine ordnungsgemäße Funktion essenziell sind.

Da der Input der Fahrzeugposition über die X- und Y-Koordinate definiert wird, jedoch openTCS nicht mit der präzisen Position des Fahrzeuges, sondern mit Knoten und Kanten arbeitet, muss eine Funktion implementiert werden, welche dem Fahrzeug ein Knoten zuweist. Dies ist z. B. notwendig, wenn das Fahrzeug nach einem Handbetrieb auf einer willkürlichen Position abgestellt wird. Infolgedessen muss openTCS mitgeteilt werden, welcher Knoten als Startknoten für einen zukünftigen Fahrbefehl referenziert werden soll. Die Lösung erfolgt über die Definition des Keys „*InitialPosition*“ im Anlagenlayout unter Fahrzeugeigenschaften. Als Wert muss einmalig die Startposition des Fahrzeuges angegeben werden, dargestellt in Abbildung 4.37. Somit ist es gleichzeitig möglich den Startpunkt des AGV‘s variabel zu setzen.

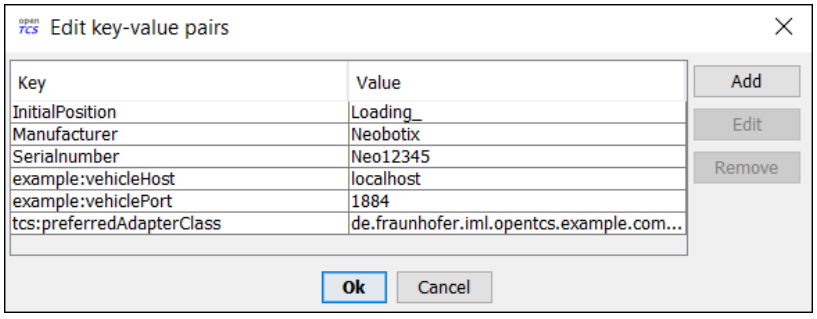

Abbildung 4.37: Initialisieren des Startpunktes, Eingabe von Herstellername und Seriennummer des Fahrzeuges über die Fahrzeugeigenschaften

Weiters können Herstellername und Seriennummer des Fahrzeuges in den Fahrzeugeigenschaften festgelegt werden. Diese Angaben werden im weiteren Verlauf für die Erstellung des Fahrbefehls nach der Vorgabe der VDA 5050 benötigt. Bei Fehlen dieser Angabe werden beide Werte auf *„Not defined“* gesetzt. Damit der eingegebene Wert ausgelesen werden kann, wurde in der Klasse „*NeobotixCommAdapterFactory*“ die Funktion *„getAdapterfor()“* mit den Code in Abbildung 4.38 erweitert. Der eingegeben Wert für „*InitialPosition“ wird* dort direkt als Startknoten des Fahrzeuges gesetzt.

```
1    if (vehicle.getProperty("Serialnumber") == null){
2      serialNumber = "Not Defined";
3    }
4    else {
5      serialNumber = vehicle.getProperty("Serialnumber");
6    }
7    adapter.getProcessModel().setVehicleProperty("SerialNumber", serialNumber);
8    adapter.setVehicleSerialNumber(serialNumber);
9
10   if (vehicle.getProperty("Manufacturer") == null){
11     manufacturer = "Not Defined";
12   }
13   else {
14     manufacturer = vehicle.getProperty("Manufacturer");
15   }
16   adapter.getProcessModel().setVehicleProperty("Manufacturer", manufacturer);
17   adapter.setVehicleManufacturer(manufacturer);
18
19   topicPrefix = vehicle.getProperty("TopicPrefix");
20   //adapter.getProcessModel().setVehicleProperty("TopicPrefix", topicPrefix);
21   adapter.setTopicPrefix(topicPrefix);
22
23   initialPosition = vehicle.getProperty("InitialPosition");
24   adapter.getProcessModel().setVehiclePosition(initialPosition);
```

Abbildung 4.38: Code zum Einlesen der Fahrzeugeigenschaften

Ist der Startknoten einmal initialisiert, erfolgt die Zuweisung neuer Knoten automatisch über die Funktion „*updatePointPosition()“*. In dieser Funktion wurde eine erlaubte Abweichung von 10 mm definiert. Ist die Distanz zwischen dem Fahrzeug und einem anfahrenden Knoten kleiner gleich der Abweichung, so wird der Knoten dem Fahrzeug zugewiesen. Infolgedessen kann der nächste Fahrbefehl mit dem Destination-Node vom Kernel gesendet werden. Ein weiterer Aspekt, welcher berücksichtigt werden muss, ist der Beladungszustand des Fahrzeuges. So soll das Fahrzeug keine Beladungsoperationen durchführen, wenn das Fahrzeug bereits beladen ist. Selbiges gilt für das Entladen des Fahrzeuges. Dies wurde in der Funktion „*canProcess()“* umgesetzt (siehe Anhang).

4.4.3 Publishen der Fahrbefehle an den Broker

Die vom Kernel erstellten Fahrbefehle werden nicht direkt zum Fahrzeug gesendet, sondern laufen über einen MQTT Broker. Somit ist es möglich, wie in Abbildung 4.39 dargestellt, dass der Broker die Fahrbefehle entweder direkt an das Fahrzeug oder indirekt an die proprietäre Leitsteuerung des Fahrzeuges weiterleitet. Diese überträgt anschließend die Befehle an das Fahrzeug.

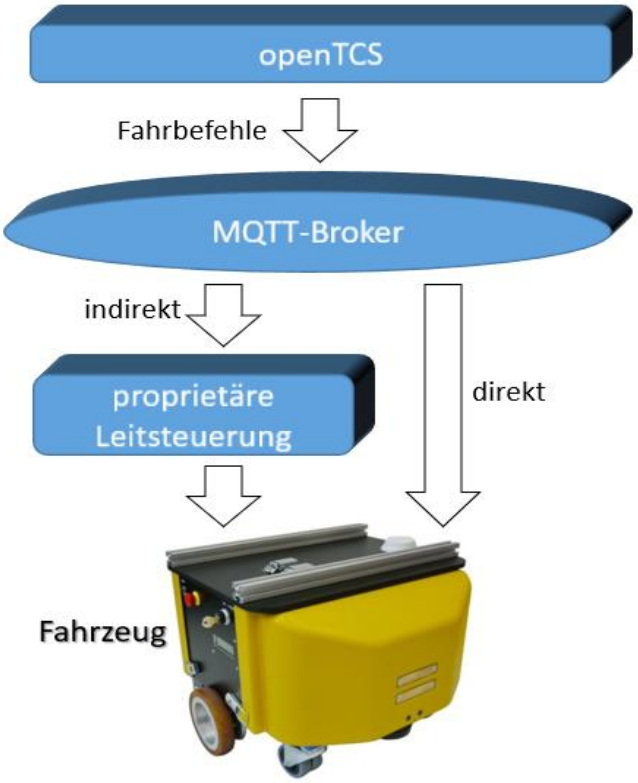

Abbildung 4.39: Einbindung der proprietären Leitsteuerung in das FTS-Bestandssystem

Die IP-Adresse und der Port des Brokers bleiben unverändert, sodass lediglich das Topic zum Publishen definiert werden muss. Hier wurde dasselbe Prinzip wie beim Abonnieren verwendet. Das Topic-prefix wird lediglich um ein Topic-level „order“ erweitert. Das Topic zum Publishen der Fahrbefehle kann dementsprechend wie folgt aufgebaut werden:

- **`Topic-Prefix/order`**
  - `Bsp.: Neobotix/Neobotix1/order`
  - `Bsp.: Pilotfabrik/Neobotix/Neobotix2/order`
  - `Bsp.: Omikron/Fahrzeug2/order`

Wie in Kapitel 3.3 erwähnt wird in der VDA 5050 die Kommunikation zwischen Fahrzeug und Leitsteuerung vorgeschrieben. Daher wurde der Fahrbefehl von openTCS nach Vorlage der 5050 umgeschrieben und in ein JSON-Format umgewandelt. Für eine detaillierte Beschreibung der Struktur und der verwendeten Datentypen wird auf die VDA5050 [53] verwiesen. Es sei angemerkt, dass alle notwendigen Datentypen aufgenommen wurden. Optionale Variablen wurden bis auf die Action-Parameter nicht berücksichtigt. Der Fahrbefehl besteht grundsätzlich aus einem Header, Informationen zum Auftrag wie orderId und den abzufahrenden Knoten und Kanten. In Abbildung 4.40 wird eine Publish-Message dargestellt, welche von openTCS an den Broker versendet wird.

Abbildung 4.40: gesendeter Fahrbefehl von openTCS an den Broker als JSON

Prinzipiell ist es möglich den Fahrbefehl als Teilauftrag zu versenden, sodass erst beim Erreichen des ersten Knotens der nächste Knoten gesendet wird. Dies führt jedoch zu einer Verzögerung des Auftrages, da das Fahrzeug stillsteht und auf den nächsten Fahrbefehl wartet. Infolgedessen wurde der Fahrbefehl als ein Ganzes gesendet. Die Beschreibung der Knoten und Kanten aus dem Fahrbefehl wird in Abbildung 4.41 dargestellt.

Abbildung 4.41: Beschreibung der Knoten und Kanten des Fahrbefehls nach Vorlage der VDA 5050

Lediglich der Action-Parameter, welcher die durchzuführende Operation am Zielpunkt beschreibt, wird erst beim Erreichen des Zielpunktes an den Broker gesendet (siehe Abbildung 4.42). Als Operations-Parameter wurde das Be- und Entladen definiert. Ein Hinzufügen von weiteren Operationen ist über den Adapter möglich und muss nachträglich hinzugefügt werden.

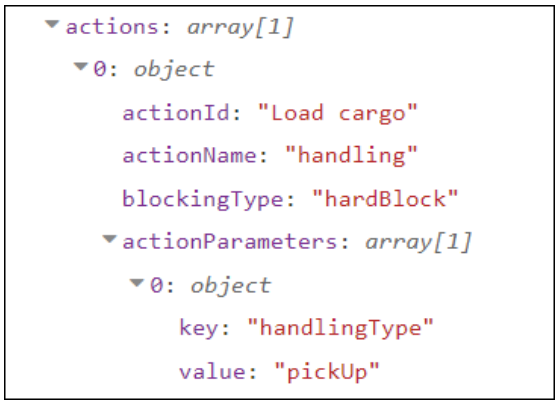

Abbildung 4.42: Beschreibung der Operationen des Fahrbefehls nach der Vorlage der VDA 5050

Der mobile Roboter bzw. dessen Steuerungssystem müssten für den praktischen Einsatz das Topic für die Fahrbefehle abonnieren und die Daten aus der JSON-Nachricht verarbeiten. Es sei weiters angemerkt, dass das Fahrzeug nicht exakt die Kanten

durchfahren muss, die im Fahrbefehl enthalten sind. Somit kann die Fahrzeugsteuerung alternative Routen einschlagen. Jedoch ist es notwendig, dass die im Fahrbefehl festgelegten Knoten erreicht werden müssen. Ein „auslassen“ von Zwischenknoten würde unweigerlich ein Fehler in openTCS verursachen.

4.4.4 Rückmeldung der Transportaufträge an den Broker

Wie o. g. ist das Erzeugen eines Transportauftrages über die Web-API problemlos möglich. Jedoch erfolgt keine Rückmeldung seitens openTCS an das MES, welche den Status der Transportaufträge wiedergibt. Infolgedessen wurde ein Code programmiert, welcher den Status der Transportaufträge eines Fahrzeuges aus openTCS ausliest und diese, bei einer Änderungen des Zustandes, an den MQTT Broker übermittelt. Der erstellte Code ist in Abbildung 4.43 dargestellt und wird im Folgenden kurz erläutert.

```
1   public void sendGetRequest() {
2       ArrayList<String> orders = new ArrayList<String>();
3       //the first entry must be initialized for comparison
4       orders.add("initialisieren");
5       try {
6           //URL of the Web-API from openTCS
7           //TODO enter the correct url of the Kernel
8           URL url = new URL("http://localhost:55200/v1/" +
9                   "transportOrders?intendedVehicle="
10                  + adapter.getProcessModel().getName());
11          HttpURLConnection httpURLConnection =
12                  (HttpURLConnection) url.openConnection();
13          httpURLConnection.setRequestMethod("GET");
14
15          //read in the Response of the server
16          String line = "";
17          InputStreamReader inputStreamReader =
18                  new InputStreamReader(httpURLConnection.getInputStream());
19          BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
20          StringBuilder response = new StringBuilder();
21          while ((line = bufferedReader.readLine()) != null) {
22              response.append(line);
23          }
24          bufferedReader.close();
25          String orderResponse = response.toString();
26          orders.add(orderResponse);
27
28          //compare Response to avoid duplication
29          for (int i = 0; i < orders.size(); i++) {
30              Object orderOld = orders.get(i);
31              for (int j = i + 1; j < orders.size(); j++) {
32                  Object orderNew = orders.get(j);
33                  //Publish Response
34                  if (!orderOld.equals(orderNew)) {
35                      System.out.println("Order new: " + orderNew.toString());
36                      PublishRequest publishRequest =
37                              new PublishRequest(adapter, orderNew);
38                      publishRequest.publish();
39                  }
40              }
41          }
42          orders.remove(0);
```

Abbildung 4.43: Code zum einlesen der Transportaufträge

Über die Web-API Schnittstelle wird über einen erstellten Client, mittels der GET-Methode, auf die Daten zugegriffen. Die Response-Nachricht wird anschließend als String in eine Arraylist hinzugefügt. Aus Versuchen wurde erkannt, dass die Response Nachricht mehrfach als Duplikat eingelesen wird. Dies wurde über Schleifen gelöst, welche die empfangenen Response-Nachrichten miteinander vergleicht und Duplikate entfernt. Stimmen die Response-Nachrichten nicht überein, wird der abgerufene Status an den MQTT Broker übermittelt. Der Code dazu befindet sich im Anhang. Als Topic wurde wie vorhin das gleiche Schema verwendet und wird wie folgt aufgebaut werden:

- **`Topic-Prefix/orderUpdate`**
  - `Bsp.: Neobotix/Neobotix1/orderUpdate`
  - `Bsp.: Pilotfabrik/Neobotix2/orderUpdate`
  - `Bsp.: Omikron/Fahrzeug2/orderUpdate`

Um die übermittelten Rückmeldungen wiedergeben und prüfen zu können, wurde mittels Node-Red ein Flow erstellt. Die empfangene JSON-Nachricht ist in Abbildung 4.44 dargestellt.

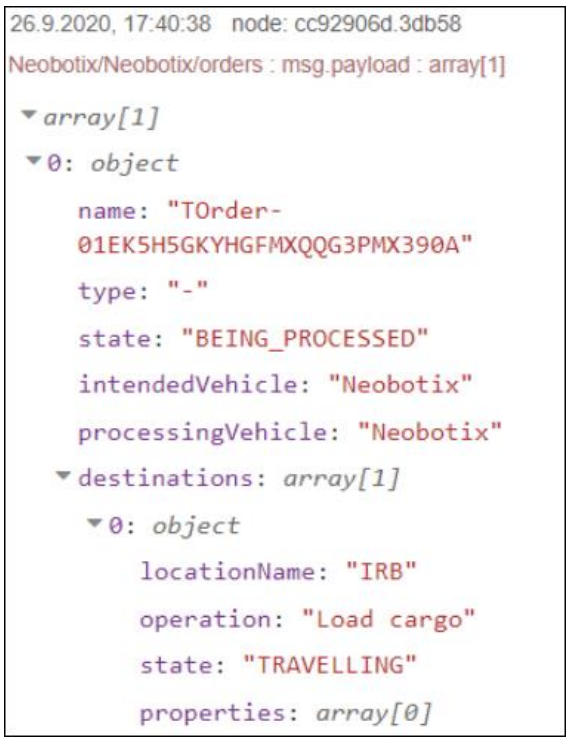

Abbildung 4.44: Rückmeldung von openTCS an den Broker bzgl. des Fortschrittes der Aufträge

5 Zusammenfassung

Der Einsatz von FTS in der diskreten Fertigung ist für produzierende Unternehmen ein aktuelles Thema. Der Trend zu hochgradig individualisierten Produkten erfordert neue Ansätze sowohl beim Planen von Produktionsstationen als auch bei der Organisation der Materialversorgung. Ziel dieser Diplomarbeit war es daher, die Implementation eines FTS innerhalb einer MES Infrastruktur zu beschreiben und Möglichkeiten zur Einbindung neuer Fahrzeuge aufzuzeigen. Aus diesem Grund wurde zuerst die Kommunikation zwischen der FTS-Leitsteuerung openTCS und dem übergeordneten Host, Node-Red, überprüft. Dabei bot openTCS eine TCP/IP und eine Web-API als Schnittstelle an.

Obwohl die TCP/IP Schnittstelle als veraltet angesehen wird, ist der Funktionsumfang in openTCS geringfügig größer als die der Web Schnittstelle. Für Unternehmen, welche noch LAN-Verbindungen nutzen, stellt diese Schnittstelle eine praktische Lösung dar. Hingegen kann gesagt werden, dass der Einsatz moderner Webtechnologien einen einfachen Betrieb in größeren Rechenzentren oder sogar einer bestehenden Cloud-Infrastruktur ermöglicht. Durch die Verwendung von bereits bestehenden und weit verbreiteten Internet-Standards wie das HTTP und JSON sind die Barrieren zum Einstieg vergleichsweise niedriger. Somit kann eine Implementierung über REST stattfinden, das standardisierte HTTP-Methoden verwendet.

Ein Problem bei openTCS war es, dass über die Host-Schnittstelle nur bestimmte Parameter übertragen werden konnten. So kann der Startzeitpunkt des Auftrages nicht angegeben werden. Infolgedessen wird der Auftrag sofort nach Eintreffen in openTCS ausgeführt. Durch eine Anpassung der Host-Schnittstelle in openTCS könnte der Satz von Parametern erweitert bzw. verändert werden, sodass zusätzliche Daten aus einem MES in openTCS eingebunden werden können.

Die Verwendung universeller Schnittstellen wie MQTT und die Protokollbeschreibung der VDA 5050 sorgen für eine Konnektivität mit unterschiedlichen Fahrzeugherstellern und verringern gleichzeitig die Implementierungszeit von neuen Fahrzeugen. Einige Anbieter, wie Incubed IT und Kinexon, haben schon darauf reagiert und bieten Flottenmanager mit diesen universellen Schnittstellen und eine Kompatibilität mit der VDA 5050 an. Jedoch gibt es Anbieter, wie AGILOX, die mit der VDA 5050 „unglücklich“ sind. Diese verwenden keinen zentralen Flottenmanager, sondern setzten auf eine dezentrale Steuerung mittels Schwarmintelligenz. Somit würde ein Befolgen von vordefinierten Routen, das Erkennen und Melden von Hindernissen und Warten auf neue Anweisungen die KI der Fahrzeuge behindern und die Leistung des Systems verringern. Der VDA 5050 Standard würde somit die Funktionalität des Systems einschränken.

Literaturverzeichnis

[1] T. Albrecht and G. Ullrich, *Fahrerlose Transportsysteme, Eine Fibel mit Praxisanwendungen zur Technik für die Planung*. Wiesbaden: Springer Vieweg, 2019.

[2] VDI-Richtlinie, "VDI 2510: Fahrerlose Transportsysteme (FTS)," *Verein Dtsch. Ingenieure, Düsseld.*, 2005, doi: 10.1007/978-3-8348-9890-6.

[3] W. Huber, *Industrie 4.0 kompakt – Wie Technologien unsere Wirtschaft und unsere Unternehmen verändern, Transformation und Veränderung des gesamten Unternehmens*. Wiesbaden: Springer Vieweg, 2018.

[4] P. Dickmann, *Schlanker Materialfluss, mit Lean Production, Kanban und Innovationen*. Heidelberg: Springer Vieweg, 2015.

[5] VDI-Richtlinie, "VDI 4451-Blatt 7: Kompatibilität von Fahrerlosen Transportsystemen (FTS)-Leitsteuerung für FTS," *Verein Dtsch. Ingenieure, Düsseld.*, 2005.

[6] T. Bauernhansl, M. ten Hompel, and B. Vogel-Heuser, *Industrie 4.0 in Produktion, Automatisierung und Logistik*. Wiesbaden: Springer Vieweg, 2014.

[7] "Tünkers AGVs," 2017. https://www.tuenkers.de/publish/binarydata/Aktuelles/2017/mai/tuenkers-agvs.pdf (accessed Mar. 25, 2020).

[8] VDI-Richtlinie, "VDI 4451-Blatt 4: Kompatibilität von Fahrerlosen Transportsystemen (FTS) Offene Steuerungsstruktur für Fahrerlose Transportfahrzeuge (FTF)," *Verein Dtsch. Ingenieure, Düsseld.*, 1998.

[9] VDI-Richtlinie, "VDI 4451-Blatt 6: Kompatibilität von Fahrerlosen Transportsystemen (FTS)- Sensorik für Navigation und Steuerung," *Verein Dtsch. Ingenieure, Düsseld.*, 2003.

[10] M. ten Hompel, T. Schmidt, and J. Dregger, *Materialflusssysteme, Förder- und Lagertechnik*. Heidelberg: Springer Vieweg, 2018.

[11] G. Ullrich, "Navigations- und Steuerungssysteme für die freie Navigation von Radfahrzeugen," Heidelberg: Springer, 1996.

[12] H. Martin, *Transport- und Lagerlogistik, Systematik, Planung, Einsatz und Wirtschaftlichkeit*. Wiesbaden: Springer Vieweg, 2004.

[13] C. Wurll, "Das bewegliche Lager auf Basis eines Cyber-physischen Systems," in *Handbuch Industrie 4.0 Bd.3*, 2017.

[14] VDI-Richtlinie, "VDI 2510 Blatt 3: Fahrerlose Transportsysteme (FTS) - Schnittstellen zu Infrastruktur und peripheren Einrichtungen," *Verein Dtsch. Ingenieure e.V., Düsseld.*, 2017.

[15] VDI-Richtlinie, "VDI 2510-Blatt 2: Fahrerlose Transportsysteme (FTS)-Sicherheit von FTS," *Verein Dtsch. Ingenieure, Düsseld.*, 2013.

[16] VDI-Richtlinie, "VDI 2510-Blatt 1: Infrastruktur und periphere Einrichtungen für Fahrerlose Transportsysteme (FTS)," *Verein Dtsch. Ingenieure e.V.*, 2009.

[17] B. Niemann, M. Baum, D.-H. Fricke, and L. Overmeyer, "Aufbau von Fahrerlosen Transportsystemen (FTS) durch eine dezentrale Datenstruktur," *Logist. J. nicht-referierte Veröffentlichungen*, 2006, doi:

10.2195/lj_not_ref_overmeyer_102006.

[18] S. Thanheiser, L. Liu, and H. Schmeck, “Selbstorganisation durch Dezentralität - Dezentralität durch Selbstorganisation: Auf dem Weg zu einem ‘organischen’ Management von Unternehmens-IT.,” 2008.

[19] D. Colling, S. Ibrahimpasic, A. Trenkle, and K. Furmans, “Dezentrale auftragserzeugung und vergabe für FTF,” *Logist. J.*, 2016, doi: 10.2195/lj_Proc_colling_de_201610_01.

[20] A. Osseiran *et al.*, “Scenarios for 5G mobile and wireless communications: The vision of the METIS project,” *IEEE Commun. Mag.*, 2014, doi: 10.1109/MCOM.2014.6815890.

[21] T. Kirk, J. Stenzel, A. Kamagaews, and T. M. Hompel, “zellulare transportfahrzeuge für flexible und wandelbare intralogistiksysteme,” *Logist. J.*, 2012, doi: 10.2195/lj_proc_kirks_de_201210_01.

[22] C. Schwarz, J. Schachmanow, J. Sauer, L. Overmeyer, and G. Ullmann, “Self guided vehicle systems,” *Logist. J.*, 2013, doi: 10.2195/lj_NotRev_schwarz_de_201312_01.

[23] R. Walenta, T. Schellekens, A. Ferrein, and S. Schiffer, “A decentralised system approach for controlling AGVs with ROS,” 2017, doi: 10.1109/AFRCON.2017.8095693.

[24] K. Feldmann and W. Wolf, “Autonom navigierende Fahrerlose Transportsysteme in der Produktion,” 2006, doi: 10.1007/3-540-30292-1_33.

[25] C. Schwarz, “Untersuchung zur Steigerbarkeit von Flexibilität, Performanz und Erweiterbarkeit von Fahrerlosen Transportsystemen durch den Einsatz dezentraler Steuerungstechniken,” Dissertation, Universität Oldenburg, Department für Informatik, 2014.

[26] A. Varal, “Konzeptionierung und Aufbau eines Testfeldes zur dezentralen Steuerung eines fahrerlosen Transportsystems,” Masterarbeit, Hochschule für Angewandte Wissenschaften Hamburg, Studiengang Produktionstechnik und -management, 2018.

[27] L. Seybold, J. Krokowicz, A. Pieczynski, and R. Setter, “Prinzipien für das Monitoring , die Planung , Regelung und Diagnose von fahrerlosen Transportsystemen,” 2011.

[28] H.-H. Wiendahl, U. Mussbach-Winter, and R. Kipp, *Marktspiegel Business Software – MES – Fertigungssteuerung*. Fraunhofer Institut Produktionstechnik und Automatisierung, Stuttgart; Trovarit AG, Aachen, 2007.

[29] K. Kurbel, *Enterprise Resource Planning und Supply Chain Management in der Industrie*. Oldenbourg: de Gruyter, 2016.

[30] J. Kletti, *MES - Manufacturing Execution System, Moderne Informationstechnologie zur Prozessfähigkeit der Wertschöpfung*. Heidelberg: Springer, 2006.

[31] VDI-Richtlinie, “VDI 5600-Blatt 1: Fertigungsmanagementsysteme, Manufacturing Execution Systems – MES,” *Verein Dtsch. Ingenieure e.V., Düsseld.*, pp. 1–78, 2016.

[32] B. Ebel, *Produktionswirtschaft, Kompendium der praktischen Betriebswirtschaft*, 9., vollst. Ludwigshafen: NWB Verlag, 2009.

[33] N. Gronau, *Enterprise Resource Planning und Supply Chain Management*. de Gruyter Oldenbourg, 2004.

[34] W. W. Osterhage, *ERP-Kompendium, Eine Evaluierung von Enterprise Resource Planning Systemen*. Heidelberg: Springer Vieweg, 2014.

[35] J. Lewandowski, “Produktionsplanung und -steuerung in mittelständischen Unternehmen unter besonderer Berücksichtigung von Manufacturing Execution Systems,” Dissertation, TU Wien, Institut für Fertigungstechnik und Hochleistungslasertechnik, 2011.

[36] J. Kletti and J. Schumacher, “Die perfekte Produktion, Manufacturing Excellence durch Short Interval Technology (SIT),” in *Die perfekte Produktion*, Heidelberg: Springer, 2011.

[37] J. Kletti, *Konzeption und Einführung von MES-Systemen*. Heidelberg: Springer, 2007.

[38] J. Kletti and R. Deisenroth, *MES-Kompendium, Ein Leitfaden am Beispiel von HYDRA*. Heidelberg: Springer Vieweg, 2019.

[39] M. Srinivasan, *Web Technology: Theory and Practice*. Boca Raton: CRC Press Taylor 6 Francis Group, 2012.

[40] O. Willem, “Gestaltung einer API zur Datenerfassung und -verwaltung mittels Eve,” Diplomarbeit, TU Wien, Forschungsbereich Maschinenbauinformatik und Virtuelle Produktentwicklung, 2017.

[41] W. Goralski, “Hypertext Transfer Protocol,” in *The Illustrated Network*, 2017.

[42] “MQTT: The Standard for IoT Messaging,” May 11, 2020. http://mqtt.org/ (accessed May 11, 2020).

[43] Dominik Obermaier, “Pub/Sub for the masses - Ein Einführungsworkshop in MQTT,” 2020. https://de.slideshare.net/dobermai/pubsub-for-the-masses-ein-einfhrungsworkshop-in-mqtt-german (accessed May 11, 2020).

[44] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” 2017, doi: IEEE International Symposium on Systems Engineering.

[45] F. Pauker, “OPC UA Information Model Design,” Dissertation, TU Wien, Institut für Fertigungstechnik und Photonische Technologien, 2019.

[46] M. Maritsch, C. Kittl, and T. Ebner, “Sichere Vernetzung von Geräten in Smart Factories mit MQTT,” *Mensch und Comput. 2015 - Work.*, pp. 217–224, 2015, doi: 10.1515/9783110443905-032.

[47] “Handbuch: TC3 IoT Communication (MQTT),” 2020. https://download.beckhoff.com/download/document/automation/twincat3/TF6701_TC3_IoT_Communication_MQTT_DE.pdf (accessed Jul. 20, 2020).

[48] A. Banks, E. Briggs, and R. Gupta, “MQTT Version 5.0 Committee Specification,” *Oasis*, no. December 2017, pp. 1–136, 2017.

[49] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, “JSON: Data model, Query languages and Schema specification,” in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2017, vol. Part F1277, pp. 123–135, doi: 10.1145/3034786.3056120.

[50] D. Crockford, “Introducing JSON.” https://www.json.org/json-en.html (accessed May 10, 2020).

[51] B. Smith, *Beginning JSON*. Heidelberg: Springer, 2015.

[52] F. Yao, A. Keller, M. Ahmad, B. Ahmad, R. Harrison, and A. W. Colombo, “Optimizing the Scheduling of Autonomous Guided Vehicle in a Manufacturing Process,” 2018, doi: 10.1109/INDIN.2018.8471979.

[53] VDA-Richtlinie, “VDA 5050: Schnittstelle zur Kommunikation zwischen Fahrerlosen Transportfahr- zeugen (FTF) und einer Leitsteuerung,” no. August, pp. 1–36, 2019.

[54] R. W. Seifert, M. G. Kay, and J. R. Wilson, “Evaluation of AGV routeing strategies using hierarchical simulation,” *Int. J. Prod. Res.*, 1998, doi: 10.1080/002075498193057.

[55] H. Sahib Hasan, M. S. Z. Abidin, M. S. A. Mahmud, and M. F. Muhamad Said, “Automated guided vehicle routing: Static, dynamic and free range,” *Int. J. Eng. Adv. Technol.*, 2019, doi: 10.35940/ijeat.E1001.0585C19.

[56] M. S. Akturk and H. Yilmaz, “Scheduling of automated guided vehicles in a decision making hierarchy,” *Int. J. Prod. Res.*, 1996, doi: 10.1080/00207549608904920.

[57] Le-Ahn T., “Intelligent Control of Vehicle-Based Internal Transport Systems.,” *ERIM PhD Ser Res Manag 51 Erasmus Univ Rotterdam*, 2005.

[58] H. Munz, G. Stöger, and J. Schreier, “OPC UA over TSN: Architektur und Anwendung.” https://www.industry-of-things.de/opc-ua-over-tsn-architektur-und-anwendung-a-927023/ (accessed Jul. 10, 2020).

[59] “openTCS: users-guide.” https://www.opentcs.org/docs/4.18/user/opentcs-users-guide.html (accessed Apr. 10, 2020).

[60] “openTCS: Structure of the software.” https://www.opentcs.org/en/structure.html (accessed Apr. 03, 2020).

[61] “KINEXON FTS-Leitsteuerung - Steuerung einer Flotte von FTS | KINEXON.” https://kinexon.com/de/technologie/fts-leitsteuerung (accessed Oct. 13, 2020).

[62] “Fleet Management – incubed IT.” https://www.incubedit.com/product/fleet-management-server/ (accessed Oct. 13, 2020).

[63] M. Gourgand, S. Xiao-Chao, and N. Tchernev, “Choice of the Guide Path Layout for an AGV based Material Handling System,” *Lab. d’Infonnatique Univ. Blaise Pascal Clen. I1 631 77 AUBIERE Cedex Fr.*, 1995.

[64] T. Le-Anh and M. B. M. De Koster, “A review of design and control of automated guided vehicle systems,” *Eur. J. Oper. Res.*, 2006, doi: 10.1016/j.ejor.2005.01.036.

[65] E. G. Coffman, M. Elphick, and A. Shoshani, “System Deadlocks,” *ACM Comput. Surv.*, 1971, doi: 10.1145/356586.356588.

[66] “openTCS: developer-guide.” https://www.opentcs.org/docs/5.0/developer/developers-guide/opentcs-developers-guide.html# (accessed Apr. 12, 2020).

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang: Programmcode

Klasse: CallbackBattery

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks;

import com.google.gson.Gson;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models.Battery;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;

/**
 * Callback to receive the battery state
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class CallbackBattery implements MqttCallback {

    private NeobotixCommAdapter adapter;
    private Gson gson;

    public CallbackBattery(NeobotixCommAdapter adapter){
        this.adapter = adapter;
        this.gson = new Gson();
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        String s = new String(message.getPayload());
        Battery voltage = gson.fromJson(s, Battery.class);
        int battery = (int) voltage.getValue();
        adapter.getProcessModel().setVehicleEnergyLevel(battery);
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {}

    @Override
    public void connectionLost(Throwable cause) {}
}
```

Klasse: CallbackLoadState

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks;

import com.google.gson.Gson;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models.LoadStateAgv;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;

/**
 * Callback to receive the Load state
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class CallbackLoadState implements MqttCallback {

    private NeobotixCommAdapter adapter;
    private Gson gson;

    public CallbackLoadState(NeobotixCommAdapter adapter){
        this.adapter = adapter;
        this.gson = new Gson();
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        String s = new String (message.getPayload());
        LoadStateAgv loadStateAgv = gson.fromJson(s, LoadStateAgv.class);
        boolean loadState = loadStateAgv.isValue();
        adapter.changeLoadState(loadState);
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {}

    @Override
    public void connectionLost(Throwable cause) {}
}
```

Klasse: CallbackOrientationAngle

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks;

import com.google.gson.Gson;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models.OrientationAngle;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;

/**
 * Callback to receive the orientation angle
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class CallbackOrientationAngle implements MqttCallback {

    private NeobotixCommAdapter adapter;
    private Gson gson;

    public CallbackOrientationAngle(NeobotixCommAdapter adapter){
        this.adapter = adapter;
        this.gson = new Gson();
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        String s = new String( message.getPayload());
        OrientationAngle steeringAngle = gson.fromJson(s, OrientationAngle.class);
        double angle = steeringAngle.getValue();
        adapter.getProcessModel().setVehicleOrientationAngle(angle);
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {}

    @Override
    public void connectionLost(Throwable cause) {}
}
```

Klasse: CallbackPositionX

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks;

import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import com.google.gson.Gson;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models.PositionX;
import org.opentcs.data.model.Triple;

/**
 * Callback to receive the x-coordinate
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class CallbackPositionX implements MqttCallback {

    private NeobotixCommAdapter adapter;
    private Gson gson;

    public CallbackPositionX(NeobotixCommAdapter adapter){
        this.adapter = adapter;
        this.gson = new Gson();
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        String s = new String(message.getPayload());
        PositionX position = gson.fromJson(s, PositionX.class);
        long positionX = position.getValue();
        Triple cPos = adapter.getPosition();
        adapter.changePosition(positionX, cPos.getY());
    }

    @Override
    public void connectionLost(Throwable cause) {}

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {}
}
```

Klasse: CallbackYPoisition

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks;

import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models.PositionY;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import com.google.gson.Gson;
import org.opentcs.data.model.Triple;

/**
 * Callback to receive the y-coordinate
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class CallbackPositionY implements MqttCallback {

    private NeobotixCommAdapter adapter;
    private Gson gson;

    public CallbackPositionY(NeobotixCommAdapter adapter){
        this.adapter = adapter;
        this.gson = new Gson();
    }

    @Override
    public void messageArrived(String topic, MqttMessage message) throws Exception {
        String s = new String(message.getPayload());
        PositionY position = gson.fromJson(s, PositionY.class);
        long positionY = position.getValue();
        Triple cPos = adapter.getPosition();
        adapter.changePosition(cPos.getX(), positionY );
    }

    @Override
    public void deliveryComplete(IMqttDeliveryToken token) {}

    @Override
    public void connectionLost(Throwable cause) {}
}
```

Klasse: Battery

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class Battery {

    private String id;
    private String path;
    private String name;
    private String timestamp;
    private double value;
    private BatteryMeta meta;
    private String source;
    private String description;

    public String getId() { return id; }

    public void setId(String id) { this.id = id;}

    public String getPath() {return path;}

    public void setPath(String path) {this.path = path; }

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public String getTimestamp() {return timestamp;}

    public void setTimestamp(String timestamp) {this.timestamp = timestamp;}

    public double getValue() {return value;}

    public void setValue(double value) {this.value = value;}

    public BatteryMeta getMeta() { return meta;}

    public void setMeta(BatteryMeta meta) { this.meta = meta; }

    public String getSource() { return source;}

    public void setSource(String source) {this.source = source;}

    public String getDescription() {return description;}

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Klasse: LoadStateAgv

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class LoadStateAgv {

    private String id;
    private String path;
    private String name;
    private String timestamp;
    private boolean value;
    private LoadStateMeta meta;
    private String source;
    private String description;

    public String getId() { return id; }

    public void setId(String id) { this.id = id;}

    public String getPath() {return path;}

    public void setPath(String path) {this.path = path; }

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public String getTimestamp() {return timestamp;}

    public void setTimestamp(String timestamp) {this.timestamp = timestamp;}

    public boolean isValue() {return value;}

    public void setValue(boolean value) {this.value = value;}

    public LoadStateMeta getMeta() { return meta;}

    public void setMeta(LoadStateMeta meta) { this.meta = meta; }

    public String getSource() { return source;}

    public void setSource(String source) {this.source = source;}

    public String getDescription() {return description;}

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Klasse: OrientationAngle

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class OrientationAngle {

    private String id;
    private String path;
    private String name;
    private String timestamp;
    private double value;
    private OrientationAngleMeta meta;
    private String source;
    private String description;

    public String getId() {return id;}

    public void setId(String id) {this.id = id;}

    public String getPath() {return path;}

    public void setPath(String path) {this.path = path;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public String getTimestamp() {return timestamp;}

    public void setTimestamp(String timestamp) {this.timestamp = timestamp;}

    public double getValue() {return value;}

    public void setValue(double value) {this.value = value;}

    public OrientationAngleMeta getMeta() {return meta;}

    public void setMeta(OrientationAngleMeta meta) {this.meta = meta;}

    public String getSource() {return source;}

    public void setSource(String source) {this.source = source;}

    public String getDescription() {return description;}

    public void setDescription(String description) {this.description = description;}
}
```

Klasse: PositionX

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class PositionX {

    private String id;
    private String path;
    private String name;
    private String timestamp;
    private long value;
    private PositionXMeta meta;
    private String source;
    private String description;

    public String getId() {return id;}

    public void setId(String id) {this.id = id;}

    public String getPath() {return path;}

    public void setPath(String path) {this.path = path;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public String getTimestamp() {return timestamp;}

    public void setTimestamp(String timestamp) {this.timestamp = timestamp;}

    public long getValue() {return value;}

    public void setValue(long value) {this.value = value;}

    public PositionXMeta getMeta() {return meta;}

    public void setMeta(PositionXMeta meta) {this.meta = meta;}

    public String getSource() {return source;}

    public void setSource(String source) {this.source = source;}

    public String getDescription() {return description;}

    public void setDescription(String description) {this.description = description;}
}
```

Klasse: PositionY

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class PositionY {

    private String id;
    private String path;
    private String name;
    private String timestamp;
    private long value;
    private PositionYMeta meta;
    private String source;
    private String description;

    public String getId() {return id;}

    public void setId(String id) {this.id = id;}

    public String getPath() {return path;}

    public void setPath(String path) {this.path = path;}

    public String getName() {return name;}

    public void setName(String name) {this.name = name;}

    public String getTimestamp() {return timestamp;}

    public void setTimestamp(String timestamp) {this.timestamp = timestamp;}

    public long getValue() {return value;}

    public void setValue(long value) {this.value = value;}

    public PositionYMeta getMeta() {return meta;}

    public void setMeta(PositionYMeta meta) {this.meta = meta;}

    public String getSource() {return source;}

    public void setSource(String source) {this.source = source;}

    public String getDescription() {return description;}

    public void setDescription(String description) {this.description = description;}
}
```

Klasse: BatteryMeta

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class BatteryMeta {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Klasse: LoadStateAgvMeta

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class LoadStateAgvMeta {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Klasse: OrientationAngleMeta

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class OrientationAngleMeta {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Klasse: PositionXMeta

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class PositionXMEta {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Klasse: PositionYMeta

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.models;

/**
 * class for assigning the variables from JSON input
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class PositionYMeta {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Klasse: ActionParameter

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

/**
 * Action-Parameters the AGV can perform, based on VDA 5050
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class ActionParameter {

    //e.g. handlingType
    private String key;

    //e.g. pickUp, dropOff
    private String value;

    public ActionParameter(String key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

Klasse: Action

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

import java.util.List;

/**
 * Action the AGV can perform; based on VDA5050
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class Action {

    //Unique action identification
    private String actionId;

    //Name of action (see List of actions) e.g. lift
    private String actionName;

    //Enum {noBlock, softBlock, hardBlock}
    private String blockingType;

    //Array of actionParameter-objects for the indicated action
    private List<ActionParameter> actionParameters;

    public Action(String actionId, String actionName, String blockingType,
                  List<ActionParameter> actionParameters) {
        this.actionId = actionId;
        this.actionName = actionName;
        this.blockingType = blockingType;
        this.actionParameters = actionParameters;
    }
}
```

Klasse: NodePosition

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

/**
 * Description of Node-Position according to vda 5050
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class NodePosition {

    //X-position on the map in reference to the world coordinate system
    private long x;

    //Y-position on the map in reference to the world coordinate system
    private long y;

    //The angular dimension
    private double theta;

    //Unique identification of the map in which the position is referenced.
    private String mapId;

    public NodePosition(long x, long y, double theta, String mapId) {
        this.x = x;
        this.y = y;
        this.theta = theta;
        this.mapId = mapId;
    }
}
```

Klasse: Edge

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

/**
 * Description of direct connection of two nodes (Edges) according to vda 5050
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class Edge {

    //Unique edge identification
    private String edgeId;

    //Id to track the sequence of nodes and
    //edges in an order and to simplify order updates
    private int sequenceId;

    //If true, the edge is part of the base plan.
    //If false, the edge is part of the horizon plan.
    private boolean released;

    //The nodeId of the start-node.
    private String startNode;

    //The nodeId of the end-node.
    private String endNode;

    public Edge(String edgeId, int sequenceId, boolean released,
                String startNode, String endNode) {
        this.edgeId = edgeId;
        this.sequenceId = sequenceId;
        this.released = released;
        this.startNode = startNode;
        this.endNode = endNode;
    }
}
```

Klasse: Node

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

import java.util.List;

/**
 * Description of Character string of nodes to be processed
 * to fulfill the order according to vda 5050
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class Node {

    //Unique node identification
    private String nodeId;

    //Id to track the sequence of nodes and edges in an
    // order and to simplify order updates
    private int sequenceId;

    //If true, the edge is part of the base plan.
    // If false, the edge is part of the horizon plan
    private boolean released;

    // JSON-Object; Defines the position on a map in world coordinates
    private NodePosition nodePosition;

    //Indicates how exact an AGV has to drive over
    // a node in order for it to count as traversed.
    private double allowedDeviation;

    //Array of actions that are to be executed on the node
    private List<Action>  actions;

    public Node(String nodeId, int sequenceId, boolean released,
                NodePosition nodePosition,
                double allowedDeviation, List<Action> actions) {
        this.nodeId = nodeId;
        this.sequenceId = sequenceId;
        this.released = released;
        this.nodePosition = nodePosition;
        this.allowedDeviation = allowedDeviation;
        this.actions = actions;
    }
}
```

Klasse: Order

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

import java.util.List;

/**
 * Communication of drive commands from the master control to the AGV
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class Order {

    //headerId of the message
    private int headerId;

    //Timestamp in ISO8601 format
    private  String timestamp;

    //Version of the protocol [Major].[Minor].[Patch]
    private  String version;

    //Manufacturer of the AGV
    private  String manufacturer;

    //Serial number of the AGV
    private String serialNumber;

    //Unique order Identification
    private String orderId;

    //orderUpdate identification
    private int orderUpdateId;

    //Array of nodes to be traversed for fulfilling the order
    private List<Node> nodes;

    //Array of edges to be traversed for fulfilling the order
    private List<Edge> edges;

    public Order(int headerId, String timestamp, String version,
                 String manufacturer, String serialNumber, String orderId,
                 int orderUpdateId, List<Node> nodes, List<Edge> edges) {
        this.headerId = headerId;
        this.timestamp = timestamp;
        this.version = version;
        this.manufacturer = manufacturer;
        this.serialNumber = serialNumber;
        this.orderId = orderId;
        this.orderUpdateId = orderUpdateId;
        this.nodes = nodes;
        this.edges = edges;
    }
}
```

Klasse: OrderFactory

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order;

import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import org.opentcs.data.model.Point;
import org.opentcs.data.order.Route;
import org.opentcs.drivers.vehicle.MovementCommand;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

/**
 * Communication of drive commands from the master control to the AGV
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class OrderFactory {

    private static final int majorVersion = 1;
    private static final int minorVersion = 0;
    private static final int patch = 0;
    private static final String manufacturer = "Neobotix";
    private MovementCommand moveCmd;
    private int headerId;
    private String serialNumber;
    private String orderId;
    private int updateId;
    private double allowedDeviation;

    public OrderFactory(int headerId, String serialNumber,
                        int updateId, String orderId,
                        MovementCommand moveCmd,
                        double allowedDeviation) {
        this.headerId = headerId;
        this.serialNumber = serialNumber;
        this.updateId = updateId;
        this.moveCmd = moveCmd;
        this.allowedDeviation = allowedDeviation;
        this.orderId = orderId;
    }

    public Order createOrder() {
        String timestamp = ZonedDateTime.now(ZoneOffset.UTC)
                .format(DateTimeFormatter.ISO_INSTANT);
        String version = majorVersion + "." + minorVersion + "." + patch;
        List<Node> nodes = extractNodes();
        List<Edge> edges = extractEdges();
        return new Order(this.headerId, timestamp, version, manufacturer,
                this.serialNumber, this.orderId, this.updateId, nodes, edges);
    }

```

```
56    public List<Node> extractNodes(){
57        //start with empty List<Node>
58        List<Node> nodeList = new ArrayList<>();
59        //work with current Step of order
60        Route.Step step = moveCmd.getStep();
61
62        //add Source and Destination Point of current Step
63        int destinationSequenceId = step.getRouteIndex();
64
65        //add Source Point only if there is movement
66        if(step.getSourcePoint() != null){
67            String sourceId = step.getSourcePoint().getName();
68            int sourceSequenceId = step.getRouteIndex();
69            destinationSequenceId = (step.getRouteIndex()) + 1;
70            boolean released = step.isExecutionAllowed();
71            NodePosition nodePosition = extractNodePosition(step.getSourcePoint());
72            List<Action> actions = createEmptyActions();
73            nodeList.add(new Node(sourceId, sourceSequenceId,
74                    released, nodePosition, allowedDeviation, actions));
75        }
76
77        //add Destination Point of current Step
78        String destinationId = step.getDestinationPoint().getName();
79        boolean firstStepReleased = step.isExecutionAllowed();
80        NodePosition firstDestinationPosition =
81                extractNodePosition(step.getDestinationPoint());
82        //action is only available for Destination of current Step
83        List<Action> operation = extractAction();
84        nodeList.add(new Node(destinationId, destinationSequenceId, firstStepReleased,
85                firstDestinationPosition, allowedDeviation, operation));
86
87        //add all Destination Points after current Step
88        for(Route.Step s : moveCmd.getRoute().getSteps()){
89            if(s.getRouteIndex() > moveCmd.getStep().getRouteIndex()){
90                String nodeId = s.getDestinationPoint().getName();
91                int sequenceId = s.getRouteIndex() + 1;
92                boolean released = s.isExecutionAllowed();
93                NodePosition nodePosition =
94                        extractNodePosition(s.getDestinationPoint());
95                List<Action> actions = createEmptyActions();
96                nodeList.add(new Node(nodeId, sequenceId, released,
97                        nodePosition, allowedDeviation, actions));
98            }
99        }
100       return nodeList;
101   }
```

```
102
103     private List<Edge> extractEdges() {
104         //start with empty Edge List
105         List<Edge> edgeList = new ArrayList<>();
106
107         //work with current Step of order
108         Route.Step step = moveCmd.getStep();
109
110         //add edge for current step only if there is movement
111         if(step.getSourcePoint() != null){
112             String firstEdge = step.getPath().getName();
113             int firstSequenceId = step.getRouteIndex();
114             boolean firstEdgeReleased = step.isExecutionAllowed();
115             String firstStartNode = step.getSourcePoint().getName();
116             String firstEndNode = step.getDestinationPoint().getName();
117             edgeList.add(new Edge(firstEdge, firstSequenceId,
118                     firstEdgeReleased, firstStartNode, firstEndNode));
119         }
120         //add all Edges after current Step
121         for(Route.Step s : moveCmd.getRoute().getSteps()){
122             if(s.getRouteIndex() > moveCmd.getStep().getRouteIndex()){
123                 String edgeId = s.getPath().getName();
124                 int sequenceId = s.getRouteIndex();
125                 boolean released = s.isExecutionAllowed();
126                 String startNode = s.getSourcePoint().getName();
127                 String endNode = s.getDestinationPoint().getName();
128                 edgeList.add(new Edge(edgeId, sequenceId, released,
129                         startNode, endNode));
130             }
131         }
132         return edgeList;
133     }
134
135     public NodePosition extractNodePosition(Point p){
136         long x = p.getPosition().getX();
137         long y = p.getPosition().getY();
138         double theta = p.getVehicleOrientationAngle();
139         if(Double.isNaN(theta)){
140             theta = 0.0;
141         }
142         //TODO find real MapId
143         String mapId = "Debug MapId";
144         return new NodePosition(x, y, theta, mapId);
145     }
146
147     /**
148      * prepare the Action field(JSON) of Message based on the
149      * operation String of the current Step
150      * @return empty Actionlist or Actionlist with one Action
151      */
152     public List<Action> extractAction(){
153         List<Action> actions = new ArrayList<>();
154         if(moveCmd.getOperation().equals(NeobotixCommAdapter.loadOperation)){
155             actions.add(createLoadAction());
156         }
```

```
157         else if(moveCmd.getOperation().equals(NeobotixCommAdapter.unloadOperation)){
158             actions.add(createUnloadAction());
159         }
160         else if(moveCmd.getOperation().equals(NeobotixCommAdapter.chargeOperation)){
161             actions.add(createChargeAction());
162         }
163         return actions;
164     }
165 
166     public List<Action> createEmptyActions(){
167         return new ArrayList<>();
168     }
169 
170     public Action createLoadAction(){
171         String actionId = "Load cargo";
172         String actionName = "handling";
173         List<ActionParameter> actionParameters = new ArrayList<>();
174         ActionParameter actionParameter =
175                 new ActionParameter("handlingType", "pickUp");
176         actionParameters.add(actionParameter);
177         String blockingType = "hardBlock";
178         return new Action(actionId, actionName, blockingType, actionParameters);
179     }
180 
181     public Action createUnloadAction(){
182         String actionId = "Unload cargo";
183         String actionName = "handling";
184         List<ActionParameter> actionParameters = new ArrayList<>();
185         ActionParameter actionParameter =
186                 new ActionParameter("handlingType", "dropOff");
187         actionParameters.add(actionParameter);
188         String blockingType = "hardBlock";
189         return new Action(actionId, actionName, blockingType, actionParameters);
190     }
191 
192     public Action createChargeAction(){
193         String actionId = "Charge";
194         String actionName = "chargeBattery";
195         List<ActionParameter> actionParameters = new ArrayList<>();
196         ActionParameter actionParameter =
197                 new ActionParameter("chargeCommand", "start");
198         actionParameters.add(actionParameter);
199         String blockingType = "hardBlock";
200         return new Action(actionId, actionName, blockingType, actionParameters);
201     }
202 }
```

Klasse: GetRequest

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.orderUpdate;

import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;

/**
 * call Transport orders for a vehicle from Web-API
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class GetRequest {

    private NeobotixCommAdapter adapter;

    public GetRequest(NeobotixCommAdapter adapter) {
        this.adapter = adapter;
    }

    public void sendGetRequest(String vehicleManufacturer) {
        ArrayList<String> orders = new ArrayList<String>();
        //the first entry must be initialized for comparison
        orders.add("initialisieren");
        try {
            //URL of the Web-API from openTCS
            //TODO enter the correct url of the Kernel
            URL url = new URL("http://localhost:55200/v1/" +
                    "transportOrders?intendedVehicle="
                    + adapter.getProcessModel().getName());
            HttpURLConnection httpURLConnection =
                    (HttpURLConnection) url.openConnection();
            httpURLConnection.setRequestMethod("GET");

            //read in the Response of the server
            String line = "";
            InputStreamReader inputStreamReader =
                    new InputStreamReader(httpURLConnection.getInputStream());
            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
            StringBuilder response = new StringBuilder();
            while ((line = bufferedReader.readLine()) != null) {
                response.append(line);
            }
            bufferedReader.close();
            String orderResponse = response.toString();
            orders.add(orderResponse);

            //compare Response to avoid duplication
            for (int i = 0; i < orders.size(); i++) {
                Object orderOld = orders.get(i);
                for (int j = i + 1; j < orders.size(); j++) {
```

```
59                      Object orderNew = orders.get(j);
60                      //Publish Response
61                      if (!orderOld.equals(orderNew)) {
62                          System.out.println("Order new: " + orderNew.toString());
63                          PublishRequest publishRequest =
64                                  new PublishRequest(adapter, orderNew,
65                                          vehicleManufacturer);
66                          publishRequest.publish();
67                      }
68                  }
69              }
70              orders.remove(0);
71          }
72          catch (IOException e) {
73              e.printStackTrace();
74          }
75      }
76  }
```

Klasse: PublishRequest

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle.orderUpdate;

import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.NeobotixCommAdapter;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

import java.util.UUID;

/**
 * Publish Transport orders of a vehicle to the MQTT-Broker
 *
 * @author Yilmaz Güzel (TU Wien)
 */
public class PublishRequest {

    private NeobotixCommAdapter adapter;
    private MqttClient client;
    private String clientId = UUID.randomUUID().toString();
    private Object orders;
    private String vehicleManufacturer;

    public PublishRequest(NeobotixCommAdapter adapter, Object orderNew,
                          String vehicleManufacturer) {
        this.adapter = adapter;
        this.orders = orderNew;
        this.vehicleManufacturer = vehicleManufacturer;
    }

    public void publish(){
        try{
            String brokerId = "tcp://" + adapter.getProcessModel().getVehicleHost() +
                    ":" + adapter.getProcessModel().getVehiclePort();
            String topic = vehicleManufacturer + "/" +
                    adapter.getProcessModel().getName() + "/orderUpdate";
            client = new MqttClient(brokerId, clientId);
            client.connect();
            byte[] payload = orders.toString().getBytes();
            MqttMessage message = new MqttMessage(payload);
            client.publish(topic, message);
            client.disconnect();
        }
        catch (MqttException e) {
            e.printStackTrace();
        }
    }
}
```

Klasse: NeobotixCommAdapter

```java
/**
 * Copyright (c) Fraunhofer IML
 */
package de.fraunhofer.iml.opentcs.example.commadapter.vehicle;

import com.google.gson.Gson;
import com.google.inject.assistedinject.Assisted;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.callbacks.*;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.
        exchange.NeobotixProcessModelTO;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order.Order;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.order.OrderFactory;
import de.fraunhofer.iml.opentcs.example.commadapter.vehicle.orderUpdate.GetRequest;
import de.fraunhofer.iml.opentcs.example.common.dispatching.LoadAction;
import de.fraunhofer.iml.opentcs.example.common.telegrams.*;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.javatuples.Triplet;
import org.opentcs.contrib.tcp.netty.TcpClientChannelManager;
import org.opentcs.customizations.kernel.KernelExecutor;
import org.opentcs.data.model.Triple;
import org.opentcs.data.model.Vehicle;
import org.opentcs.drivers.vehicle.BasicVehicleCommAdapter;
import org.opentcs.drivers.vehicle.MovementCommand;
import org.opentcs.drivers.vehicle.management.VehicleProcessModelTO;
import org.opentcs.util.ExplainedBoolean;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.inject.Inject;
import java.beans.PropertyChangeEvent;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;

import static de.fraunhofer.iml.opentcs.example.common.
        telegrams.BoundedCounter.UINT16_MAX_VALUE;
import static java.util.Objects.requireNonNull;

/**
 * implementation of the communication adapter via mqtt
 *
 * @author Yilmaz Güzel (Fraunhofer IML)
 */
public class NeobotixCommAdapter
    extends BasicVehicleCommAdapter {


  /**
   * This class's logger.
   */
  private static final Logger LOG = LoggerFactory.getLogger(NeobotixCommAdapter.class);
  /**
   * <topic, ClientId, Callback class> for each Attribute updated by vehicle over MQTT.
   */
  private Triplet<String,String, MqttCallback>[] callbackData;
```

```
59  /**
60   * client List for all Callbacks clients.
61   */
62  private List<MqttClient> clientList;
63  /**
64   * if working is true: command has been sent to the vehicle and the adapter
65   * waits for its completion.
66   */
67  private boolean working = false;
68  /**
69   * Movement Command for the vehicle.
70   */
71  private MovementCommand currentCmd;
72  /**
73   * Current Order-Id of the Movement.
74   */
75  private String currentOrderId;
76  /**
77   * client for publishing orders.
78   */
79  private MqttClient sendMovementClient;
80  /**
81   * max allowed distance from Point Position.
82   */
83  private final double allowedDeviation = 10.0;
84  /**
85   * define Load State of the Agv.
86   */
87  private LoadState loadState = LoadState.UNKNOWN;
88  /**
89   * define Load Operation.
90   */
91  public static final String loadOperation = "Load cargo";
92  /**
93   * define Unload Operation.
94   */
95  public static final String unloadOperation = "Unload cargo";
96  /**
97   * define Charge Operation.
98   */
99  public static final String chargeOperation = "Charge";
100 /**
101  * counter for header ID of the message.
102  */
103 private int headerId = 0;
104
105 private Gson gson = new Gson();
106 /**
107  * Unique AGV Serial Number; comes from NeobotixCommAdapterFactory class.
108  */
109 private String vehicleSerialNumber;
110 /**
111  * AGV Manufacturer Name; comes from NeobotixCommAdapterFactory class.
112  */
113 private String vehicleManufacturer;
114 /**
115  * topic prefix for mqtt
116  */
```

```
117   private String topicPrefix;
118   /**
119    * Maps movement commands from openTCS to the telegrams sent to the attached vehicle.
120    */
121   private final OrderMapper orderMapper;
122   /**
123    * The components factory.
124    */
125   private final NeobotixAdapterComponentsFactory componentsFactory;
126   /**
127    * The kernel's executor service.
128    */
129   private final ExecutorService kernelExecutor;
130   /**
131    * Manages counting the ids for all {@link Request} telegrams.
132    */
133   private final BoundedCounter globalRequestCounter =
134           new BoundedCounter(0, UINT16_MAX_VALUE);
135   /**
136    * Maps commands to order IDs so we know which command to report as finished.
137    */
138   private final Map<MovementCommand, Integer> orderIds = new ConcurrentHashMap<>();
139   /**
140    * Manages the channel to the vehicle.
141    */
142   private TcpClientChannelManager<Request, Response> vehicleChannelManager;
143   /**
144    * Matches requests to responses and holds a queue for pending requests.
145    */
146   private RequestResponseMatcher requestResponseMatcher;
147   /**
148    * A task for enqueuing state requests periodically.
149    */
150   private StateRequesterTask stateRequesterTask;
151   /**
152    * Http Request to give Feedback to the Broker about Order-States
153    */
154   private GetRequest getRequest = new GetRequest(this);
155
156   /**
157    * Creates a new instance.
158    *
159    * @param vehicle           The attached vehicle.
160    * @param orderMapper       The order mapper for movement commands.
161    * @param componentsFactory The components factory.
162    * @param kernelExecutor    The kernel's executor service.
163    */
164
165   @Inject
166   public NeobotixCommAdapter(@Assisted Vehicle vehicle,
167                              OrderMapper orderMapper,
168                              NeobotixAdapterComponentsFactory componentsFactory,
169                              @KernelExecutor ExecutorService kernelExecutor) {
170     super(new NeobotixProcessModel(vehicle), 3, 2,
171             LoadAction.CHARGE, kernelExecutor);
172     this.orderMapper = requireNonNull(orderMapper, "orderMapper");
173     this.componentsFactory = requireNonNull(componentsFactory, "componentsFactory");
174     this.kernelExecutor = requireNonNull(kernelExecutor, "kernelExecutor");
```

```
175   }
176
177
178   public void setVehicleSerialNumber(String vehicleSerialNumber) {
179     this.vehicleSerialNumber = vehicleSerialNumber;
180   }
181
182   public void setVehicleManufacturer(String vehicleManufacturer) {
183     this.vehicleManufacturer = vehicleManufacturer;
184   }
185   public void setTopicPrefix(String topicPrefix) {
186     this.topicPrefix = topicPrefix;
187   }
188
189   @Override
190   public void initialize() {
191     super.initialize();
192   }
193
194   @Override
195   public void terminate() {
196     super.terminate();
197   }
198
199   @Override
200   public synchronized void enable() {
201     if (isEnabled()) {
202       return;
203     }
204     super.enable();
205     getProcessModel().setVehicleState(Vehicle.State.IDLE);
206     getProcessModel().setVehiclePrecisePosition(new Triple(0, 0, 0));
207   }
208
209   @Override
210   public synchronized void disable() {
211     if (!isEnabled()) {
212       return;
213     }
214     super.disable();
215   }
216
217   @Override
218   public synchronized void clearCommandQueue() {
219     super.clearCommandQueue();
220   }
221
222   @Override
223   protected synchronized void connectVehicle() {
224     String brokerIp = getProcessModel().getVehicleHost();
225     int brokerPort =  getProcessModel().getVehiclePort();
226     clientList = new ArrayList<>();
227     callbackData = new Triplet[]{
228             new Triplet<String,String,MqttCallback>(topicPrefix +
229                     "/position/x", "PositionX Client",
230                     new CallbackPositionX(this)),
231             new Triplet<String,String,MqttCallback>(topicPrefix +
232                     "/position/y", "PositionY Client",
```

```
233                     new CallbackPositionY(this)),
234             new Triplet<String,String,MqttCallback>(topicPrefix +
235                     "/position/a", "Orientation Angle Client",
236                     new CallbackOrientationAngle(this)),
237             new Triplet<String,String,MqttCallback>(topicPrefix +
238                     "/battery/voltage_percentage", "Battery Client",
239                     new CallbackBattery(this)),
240             new Triplet<String,String,MqttCallback>(topicPrefix +
241                     "/controller/loading", "Load State Client",
242                     new CallbackLoadState(this)),
243     };
244     try {
245         for(Triplet<String,String,MqttCallback> t : callbackData){
246           MqttClient client = new MqttClient("tcp://"+ brokerIp
247                   + ":" + brokerPort,t.getValue1());
248           client.setCallback(t.getValue2());
249           client.connect();
250           client.subscribe(t.getValue0(),2);
251           clientList.add(client);
252         }
253         sendMovementClient = new MqttClient("tcp://" + brokerIp + ":" +
254                 getProcessModel().getVehiclePort(), "MovementClient");
255         sendMovementClient.connect();
256     }
257     catch (MqttException e) {
258       e.printStackTrace();
259     }
260   }
261
262   @Override
263   protected synchronized void disconnectVehicle() {
264     for(MqttClient client : clientList){
265       if(client != null && client.isConnected()) {
266         try {
267           client.disconnect();
268           client.close();
269         }
270         catch (MqttException e) {
271           e.printStackTrace();
272         }
273       }
274     }
275     clientList = null;
276     if(sendMovementClient != null && sendMovementClient.isConnected()) {
277       try {
278         sendMovementClient.disconnect();
279         sendMovementClient.close();
280         sendMovementClient = null;
281       }
282       catch (MqttException e) {
283         e.printStackTrace();
284       }
285     }
286   }
287
288   @Override
289   protected synchronized boolean isVehicleConnected() {
290     return sendMovementClient.isConnected();
```

```
291   }
292
293   @Override
294   public void propertyChange(PropertyChangeEvent evt) {
295     super.propertyChange(evt);
296   }
297
298   @Override
299   public final NeobotixProcessModel getProcessModel() {
300     return (NeobotixProcessModel) super.getProcessModel();
301   }
302
303   @Override
304   protected VehicleProcessModelTO createCustomTransferableProcessModel() {
305     //Add extra information of the vehicle when sending to other
306     //software like control center or
307     //plant overview
308     return new NeobotixProcessModelTO()
309             .setVehicleRef(getProcessModel().getVehicleReference())
310             .setCurrentState(getProcessModel().getCurrentState())
311             .setPreviousState(getProcessModel().getPreviousState())
312             .setLastOrderSent(getProcessModel().getLastOrderSent())
313             .setDisconnectingOnVehicleIdle(getProcessModel()
314                     .isDisconnectingOnVehicleIdle())
315             .setLoggingEnabled(getProcessModel().isLoggingEnabled())
316             .setReconnectDelay(getProcessModel().getReconnectDelay())
317             .setReconnectingOnConnectionLoss(getProcessModel()
318                     .isReconnectingOnConnectionLoss())
319             .setVehicleHost(getProcessModel().getVehicleHost())
320             .setVehicleIdle(getProcessModel().isVehicleIdle())
321             .setVehicleIdleTimeout(getProcessModel().getVehicleIdleTimeout())
322             .setVehiclePort(getProcessModel().getVehiclePort())
323             .setPeriodicStateRequestEnabled(getProcessModel()
324                     .isPeriodicStateRequestEnabled())
325             .setStateRequestInterval(getProcessModel().getStateRequestInterval());
326   }
327
328   /**
329    * after receiving an order openTCS calls  sendCommand several
330    * times at once for each step of that order
331    * @param cmd
332    * @throws IllegalArgumentException
333    */
334   @Override
335   public synchronized void sendCommand(MovementCommand cmd)
336           throws IllegalArgumentException {
337     requireNonNull(cmd, "cmd");
338     synchronized (this) {
339       if (!working) {
340         startWorking();
341         //TODO change the location of the
342         // method call or change the algorithm as required
343         getRequest.sendGetRequest(topicPrefix);
344       }
345     }
346   }
347
348   /**
```

```
349   * delay the processing of the initial order so the getSentQueue can be filled
350   */
351  public synchronized void startWorking(){
352    working = true;
353    getProcessModel().setVehicleState(Vehicle.State.EXECUTING);
354    Thread workerThread = new Thread(() -> {
355      try {
356        Thread.sleep(1000);
357      }
358      catch (InterruptedException e) {
359        e.printStackTrace();
360      }
361      currentOrderId = UUID.randomUUID().toString();
362      sendNextCmd();
363    });
364    workerThread.start();
365  }
366
367  public synchronized Triple getPosition(){
368    Triple p =  getProcessModel().getVehiclePrecisePosition();
369    return p;
370  }
371
372  public synchronized void changePosition (long x, long y){
373      getProcessModel().setVehiclePrecisePosition(new Triple(x, y, 0));
374      updateState();
375  }
376
377  /**
378   * after a Callback check if a Point has benn reached and a
379   * command has been finished.
380   */
381  public synchronized void updateState() {
382    if(currentCmd != null){
383      updatePointPosition();
384      //check if command has finished by comparing destination to current Point
385      // for movement and operation to load state for action
386      if (currentCmd.getStep().getDestinationPoint().getName().equals(getProcessModel()
387              .getVehiclePosition())
388              && (currentCmd.isWithoutOperation() ||
389              (currentCmd.getOperation().equals(loadOperation) &&
390                      loadState == LoadState.FULL) ||
391              (currentCmd.getOperation().equals(unloadOperation) &&
392                      loadState == LoadState.EMPTY))) {
393        getProcessModel().commandExecuted(currentCmd);
394        //if current Order is finished create new ID for next order.
395        if(currentCmd.isFinalMovement()){
396          //TODO change the location of the
397          // method call or change the algorithm as required
398          getRequest.sendGetRequest(topicPrefix);
399          currentOrderId = UUID.randomUUID().toString();
400        }
401        sendNextCmd();
402      }
403    }
404  }
405
406  /**
```

```
407   * publish next command in Queue if it is not empty otherwise stop working
408   */
409  public synchronized void sendNextCmd(){
410    currentCmd = getSentQueue().poll();
411    if(currentCmd == null){
412      getProcessModel().setVehicleState(Vehicle.State.IDLE);
413      currentOrderId = null;
414      working = false;
415    }
416    else{
417      publishMovementCmd(currentCmd);
418    }
419  }
420
421  public synchronized void publishMovementCmd(MovementCommand movementCommand){
422    OrderFactory orderFactory = new OrderFactory(headerId, vehicleSerialNumber,
423            vehicleManufacturer, currentCmd.getStep().getRouteIndex(),
424            currentOrderId, currentCmd, allowedDeviation);
425    Order order = orderFactory.createOrder();
426    String payload = gson.toJson(order);
427    MqttMessage message = new MqttMessage(payload.getBytes());
428    //interfaceName/majorVersion/manufacturer/serialNumber/topic
429    try {
430      sendMovementClient.publish(topicPrefix + "/order", message);
431      LOG.debug("Published Order");
432    }
433    catch (MqttException e) {
434      e.printStackTrace();
435    }
436    headerId++;
437  }
438
439  /**
440   * calculate if Destination is within allowed deviation and set
441   * Position if new Destination has reached.
442   */
443  public synchronized void updatePointPosition(){
444    Triple position = getProcessModel().getVehiclePrecisePosition();
445    Triple targetPosition = currentCmd.getStep().getDestinationPoint().getPosition();
446
447    double distance = Math.sqrt(
448            Math.pow(targetPosition.getX() - position.getX(), 2) +
449            Math.pow(targetPosition.getY() - position.getY(), 2));
450
451    if( distance <= allowedDeviation){
452      getProcessModel().setVehiclePosition(currentCmd.getStep()
453              .getDestinationPoint().getName());
454    }
455  }
456
457  public synchronized void changeLoadState(boolean full){
458    if(full){
459      loadState = LoadState.FULL;
460    }
461    else{
462      loadState = LoadState.EMPTY;
463    }
464  }
```

```
465
466   @Override
467   public synchronized ExplainedBoolean canProcess(List<String> operations) {
468     if(working){
469       return new ExplainedBoolean(false,
470               "Vehicle is processing other Orders");
471     }
472     if(loadState == LoadState.UNKNOWN){
473       return new ExplainedBoolean(false,
474               "Load State of Vehicle is not reported");
475     }
476     boolean loaded = loadState == LoadState.FULL;
477     Iterator<String> opIter = operations.iterator();
478     while ( opIter.hasNext()) {
479       final String nextOp = opIter.next();
480       // If we're loaded, we cannot load another piece, but could unload.
481       if (loaded) {
482         if (nextOp.startsWith(loadOperation)) {
483           return new ExplainedBoolean(false,
484                   " Vehicle can not load when already loaded");
485         }
486         else if (nextOp.startsWith(unloadOperation)) {
487           loaded = false;
488         }
489       }
490       // If we're not loaded, we could load, but not unload.
491       if (!loaded){
492         if (nextOp.startsWith(loadOperation)) {
493           loaded = true;
494         }
495         else if (nextOp.startsWith(unloadOperation)) {
496           return new ExplainedBoolean(false ,
497                   " Vehicle can not unload when already empty");
498         }
499       }
500     }
501     return new ExplainedBoolean(true, " Processing possible");
502   }
503
504   @Override
505   public void processMessage(Object message) {
506     //Process messages sent from the kernel or a kernel extension
507   }
508
509   private enum LoadState {
510     EMPTY,
511     FULL,
512     UNKNOWN,
513   }
514
515 }
```