

BACHELORARBEIT

Analyse der Auswirkung von Progressive Web Apps auf bestehende Web Apps

durchgeführt am
Studiengang Informationstechnik & System-Management
an der
Fachhochschule Salzburg GmbH

vorgelegt von
Refik Kerimi



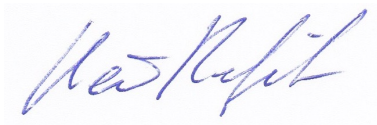
Studiengangsleiter: FH-Prof. DI Dr. Gerhard Jöchl
Betreuer: DI Norbert Egger BSc

Salzburg, September 2018

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit ohne unzulässige fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und alle aus ungedruckten Quellen, gedruckter Literatur oder aus dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte gemäß den Richtlinien wissenschaftlicher Arbeiten zitiert, bzw. mit genauer Quellenangabe kenntlich gemacht habe. Diese Arbeit wurde in gleicher oder ähnlicher Form weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt und stimmt mit der durch die Begutachter beurteilten Arbeit überein.

Salzburg, am 1.09.2018



Refik Kerimi

1410555043

Matrikelnummer

Allgemeine Informationen

Vor- und Zuname:	Refik Kerimi
Institution:	Fachhochschule Salzburg GmbH
Studiengang:	Informationstechnik & System-Management
Titel der Masterarbeit:	Analyse der Auswirkung von Progressive Web Apps auf bestehende Web Apps
Schlagwörter:	PWA, Manifest, Service Workers, Push Notification, Cach API
Betreuer an der FH:	DI Norbert Egger BSc

Kurzfassung

Die steigende Anzahl von mobilen Geräten erfordert ein Umdenken in der Art der Applikationsentwicklung.

Die steigenden Anforderungen an moderne Energiesysteme setzen eine Weiterentwicklung der Kommunikations- und Informationsinfrastruktur voraus. Das Ziel ist es, erneuerbare Energiequellen einfacher und effizienter in bestehende Netze zu integrieren. Das Projekt OpenNES, welches gemeinsam vom **AIT!** (**AIT!**), der FH Salzburg und der Fronius International GmbH entwickelt wird, soll eine offene, generische und interoperable Informations- und Automatisierungslösung schaffen, die dies ermöglicht. OpenNES stellt fern-programmierbare Funktionen, geeignete Modellierungsmethoden für Energiequellen und eine Infrastruktur zur Schaffung der Kommunikation bereit. Zur Implementierung von nicht OpenNES fähigen Geräten in Smart Grids werden mehrere Protokolladapter benötigt. Diese befinden sich im Connectivity Modul, welches die Schnittstelle zur Kommunikation nach außen darstellt. In dieser Bachelorarbeit wird der Protokolladapter für Modbus/SunSpec entwickelt. OpenNES soll dazu beitragen, dass die geforderten Ziele und Richtlinien der EU für Klimaschutz und Energie in Zukunft erreicht werden können.

Abstract

Increasing requirements on modern energy systems require further development of communication and information infrastructure. The aim is to integrate renewable energy sources more easily and efficiently into existing networks. To ensure this, the project OpenNES, which is being developed jointly by the Austrian Institute of Technology (ATI), the FH Salzburg and Fronius International GmbH, shall create an open, generic and interoperable information and automation solution. OpenNES provides remote programmable features, appropriate modelling methods for energy sources, and an infrastructure to provide communication. Several protocol adapters are required to implement non-OpenNES-enabled devices in SmartGrids. These are located within the Connectivity Module, an interface for communication to the outside. In this bachelor thesis, the protocol adapter for Modbus/SunSpec is developed. OpenNES is intended to help achieve the EU's objectives and directives for climate protection and energy in the near future.

Danksagung

Danken möchten wir vor allem unseren Betreuern für die Unterstützung bei dieser Bachelorarbeit.

Besonderer Dank gilt auch unseren Familien und Freunden, die uns während des Studiums in allen Belangen immer unterstützt haben.

Inhaltsverzeichnis

Abkürzungsverzeichnis	i
Abbildungsverzeichnis	ii
Tabellenverzeichnis	iii
Listingverzeichnis	iv
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	3
2 Progressive Web Application	4
2.1 OpenNES SmartOS	4
2.2 Basisfunktionen	5
2.3 Softwarekomponenten	5
2.4 Sicherheitsmodul	7
2.5 Virtual Functional Bus	7
2.6 Registrierung	7
2.7 Connectivity Modul	8
3 Basistechnologien	9
3.1 Modbus Protokoll	9
3.1.1 Geschichte	9
3.1.2 Grundlagen	9
3.1.3 Protokollaufbau	10
3.1.4 Modbus Datenmodell	12
3.1.5 Funktionscodes	14
3.1.6 ModBus TCP/IP	16
3.2 SunSpec	19
3.2.1 Übersicht	19
3.2.2 SunSpec Informationsmodell	19
3.2.3 SunSpec Datentypen	20
3.2.4 Register Mapping	22
4 Entwurf	23

4.1	Übersicht Adapter Pattern	23
4.2	Anforderungsanalyse	24
5	Implementierung	26
5.1	Umsetzung	26
5.2	Ausgewählte Programmiersprache und IDE	26
5.3	SunSpecDataType	26
5.4	Status	27
5.5	Remoteaddress	27
5.6	Dataobject	29
5.7	Result	29
5.8	ModbusTCPAdapter	30
5.8.1	Grundeinstellungen	30
5.8.2	ConnectionParameter	30
5.8.3	SplitString	31
5.9	Push Funktion (send)	31
5.10	Request-Response Funktion (request)	33
5.11	Publish-Subscribe Funktion (subscribe, unsubscribe)	34
5.11.1	Connectivity Modul	34
5.11.2	Subscriber	35
5.11.3	Observer	35
5.11.4	Grabber	35
6	Funktionstest/Validierung	37
6.1	Virtual Machine Management	37
6.2	Ausgangsbedingungen und Abgrenzungen	38
6.2.1	Modbus Server	38
6.3	Push1-N (send)	39
6.3.1	Ausgangssituation	39
6.3.2	Test	39
6.3.3	Testende	43
6.4	Request-Response (request)	44
6.4.1	Test	44
7	Fazit	48

Abkürzungsverzeichnis

PWA	Progressive Web Application
ADU	Application Data Unit
AGF	Advanced Grid Features
SP	Smart Phone

Abbildungsverzeichnis

2.1	Übersicht des OpenNES Konzepts [?]	5
2.2	OpenNES System Architektur [?]	6
2.3	Softwarekomponente [?]	7
2.4	Connectivity Modul [?]	8
3.1	Schichten des ModBus Protokolls [?, S.2]	10
3.2	Schematische Darstellung eines Modbus RTU Frames [?]	10
3.3	Fehlerfreie Modbus Transaktion [?, S.4]	11
3.4	Exception Antwort einer Modbus Transaktion [?, S.4]	12
3.5	Modbus Adressmodell [?, S.8]	13
3.6	Modbus ADU Vergleich [?]	16
3.7	Modbus Encapsulation bei TCP/IP [?]	17
3.8	Modbus TCP/IP Kommunikations-Übersicht [?, S.4]	18
4.1	UML Diagramm des Objekt- bzw. Klassenadapters [?, S.159]	23
4.2	Adressmapping OpenNES zu Adapter	25
6.1	Netzwerkübersicht	37
6.2	Modbus Server VM1 vor dem Test	39
6.3	Modbus Server VM2 vor dem Test	39
6.4	Modbus Server VM1 nach dem Test	43
6.5	Modbus Server VM2 nach dem Test	43

Tabellenverzeichnis

3.1	Modbus Datenmodell [?]	12
3.2	Anforderung Lese Holding Registers [?]	14
3.3	Antwort Lese Holding Registers wenn erfolgreich [?]	14
3.4	Antwort Lese Holding Registers wenn Fehler [?]	14
3.5	Anforderung Schreibe Multiple Registers [?]	15
3.6	Antwort Schreibe Multiple Registers wenn erfolgreich [?]	15
3.7	Antwort Schreibe Multiple Registers wenn Fehler [?]	15
3.8	16 bit Integer [? , S.14]	20
3.9	String [? , S.15]	21
3.10	Float [? , S.16]	21
5.1	Komponenten der Remoteaddress	27
5.2	Grundeinstellungen Modbus TCP Adapter	30
6.1	Übersicht VMs	37
6.2	Modbus Request der Push Funktion	41
6.3	Modbus Response der Push Funktion	42
6.4	Fehlerhafte Modbus Response der Push Funktion	43
6.5	Modbus Request der Request-Response Funktion	45
6.6	Modbus Response der Request-Response Funktion	46
6.7	Fehlerhafte Modbus Response der Request-Response Funktion	47

Listings

5.1	SunSpecDataType	26
5.2	Status	27
5.3	Zusammensetzung Remoteaddress	28
5.4	Dataobject	29
5.5	Result	29
5.6	Push 1:n	31
5.7	Request-Response	33
5.8	subscribe	34
5.9	unsubscribe	34
5.10	Connectivity Modul als enum Singleton	35
6.1	Methodenaufruf send	40
6.2	Methodenaufruf request	44

1 Einleitung

Durch die Markteinführung des Smart Phone hat sich in unserem Leben einiges geändert. Nicht nur unsere Kommunikation wurde dadurch verändert sondern auch unser Leben im allgemeinen, ist durch dieses kleine Wundergerät verändert worden. Wir haben fast ständig das SP im Einsatz um zu organisieren, spielen, Musik hören, Kontakte pflegen und ab und zu wird es auch zum telefonieren verwendet. Das Smart Phone hat nicht nur unser Leben verändert sondern auch das Internet und die Entwicklung von Webapplikationen. Kurz nach der Erfindung des smarten Handys kam ein weiterer Markt hinzu der sich parallel dazu entwickelt hat und es wurden neue Berufe gegründet wie der NativeApp Entwickler.

Der steigende Energieverbrauch, der sich laut Statistik Austria (Stand 2016) seit den 70er Jahren fast verdoppelt hat, erfordert ein Umdenken in der Planung und Bereitstellung verschiedenster Energiequellen [?]. Die Vernetzung und Digitalisierung unserer Welt lassen den Stromverbrauch immer rasanter steigen. Neue und nachhaltige Stromerzeuger, wie z.B.: Photovoltaikanlagen, werden in unserer Gesellschaft immer wichtiger. Ein Problem stellt allerdings die Zusammenführung Intelligenter und Simpler Energiegeräte dar. Die bestehende Infrastruktur reicht nicht aus, um Energiesysteme großflächig und gezielt steuern zu können. Diese steigenden Anforderungen setzen eine Weiterentwicklung der Kommunikations- und Informationsinfrastruktur voraus.

1.1 Motivation

Die zunehmende Anzahl an dezentralen Stromerzeugungssystemen (engl. **DER!**, **DER!**) führt zu einem Umdenken bei der Planung und beim Betrieb von elektrischen Anlagen. Intelligente Stromnetze (engl. Smart Grids) sollen den Wandel von einem klassischen Verteilernetz, mit wenigen zentralen Großerzeugern, zu einem Netz mit vielen dezentralen Erzeugern ermöglichen und dadurch zu einer effizienteren Nutzung der Stromnetzinfrasturktur beitragen [? ?].

Zurzeit besteht noch kein einheitliches System zur Integration von **DER!** Steuerungen in dem Smart Grid Bereich, da die Skalierbarkeit und Offenheit in den benötigten Automatisierungssystemen fehlt. Ein weiteres Problem stellen die vielen unterschiedlichen Kommunikationsprotokolle dar. Um Automatisierungs- und Steuerungsaufgaben in Smart Grids einfach und wiederverwendbar einzugliedern wird von **AIT!** (**AIT!**), FH Salzburg und Fronius International GmbH OpenNES entwickelt. OpenNES soll eine Informations- und Automatisierungslösung schaffen, die offen, generisch und inte-

roperabel zur vorhandenen Infrastruktur ist und zusätzlich den aktuellen Qualitätsanforderungen hinsichtlich entspricht.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, einen **PA!** für die Kommunikation zwischen OpenNES fähigen **IED! (IED!)**s und **SED! (SED!)**s, welche via Modbus/SunSpec kommunizieren, zu entwickeln und diesen in das Connectivity Modul zu integrieren. Mit Hilfe des **PA!**s soll ein einheitliches, flexibles System zur Integration von bestehenden Standards (z.B.: IEC 61850, ModBus,...) in OpenNES geschaffen werden. Weiterführend soll erreicht werden, dass durch die Verwendung von offenen und generischen Systemen, wie OpenNES, die Netzbetreiber eine gezielte und aktive Unterstützung bekommen um Systeme und den Stromfluss großflächig zu beeinflussen.

2 Progressive Web Application

Wie in Kapitel 1 beschrieben, hat der stetige Zuwachs von PWAs zum Umdenken bei der Planung und beim Entwickeln von Webapplikationen geführt [?]. Diese Arbeit beschäftigt sich mit der Frage "Können Progressive Web Apps Native Apps zur Gänze ersetzen?".

Wie in Kapitel 1 beschrieben, hat der stetige Zuwachs von **DER!**-Systemen zum Umdenken bei der Planung und beim Betreiben von Stromnetzen geführt [?]. Das Konzept der **SGSY!** (**SGSY!**) ermöglicht die Verwendung von existierenden Anlagen auf eine effizientere Art und Weise und trägt damit zu einem höheren Marktanteil von DER Systemen bei. Solche Ansätze erfordern neue Lösungen in der **IKT!**, Automatisierungsarchitektur und in der Steuerung der Systeme. OpenNES umfasst das Konzept, wie in [?] beschrieben und in Abbildung 2.1 dargestellt, einer **IKT!**-Lösung zur Integration von erneuerbaren Energiequellen in **SGSY!** [?]. Die Entwicklung beschäftigt sich mit fernprogrammierbaren Funktionen, generischen Kommunikationsstrukturen sowie dazugehörigen Anwendungsmodellierungsmethoden für **DER!** Komponenten. Einen weiteren Schwerpunkt von OpenNES stellt der Zugriff auf die **DER!** Systeme dar, wobei die unterschiedlichen Benutzerrollen berücksichtigt werden müssen. OpenNES nimmt im Bereich der **SGSY!** eine zentrale Rolle ein, da es eine Open Source Plattform bietet.

2.1 OpenNES SmartOS

Das OpenNES Smart OS ist die Softwareplattform von OpenNES [?]. Es ist eine Plattform zur Steuerung von **DER!** Komponenten, mit dem Ziel die Flexibilität, die Erweiterungsmöglichkeiten und die Kompatibilität zu erhöhen. In Abbildung 2.2 ist das SmartOS von OpenNES zu sehen. Die Steuerungsplattform beinhaltet drei Hauptkomponenten:

- Das Betriebssystem mit den Basis- und Kommunikationsfunktionen
- Das Sicherheitsmodul und austauschbare **SWK!** (**SWK!**)
- Die Entwicklungsumgebung um die **SWK!** zu programmieren oder neu zu konfigurieren

Das SmartOS ist kein normales Betriebssystem, sondern es besteht aus verschiedenen Abstraktionsschichten mit zusätzlicher Funktionalität zum Behandeln der Softwarekomponenten und dessen Interaktionen.

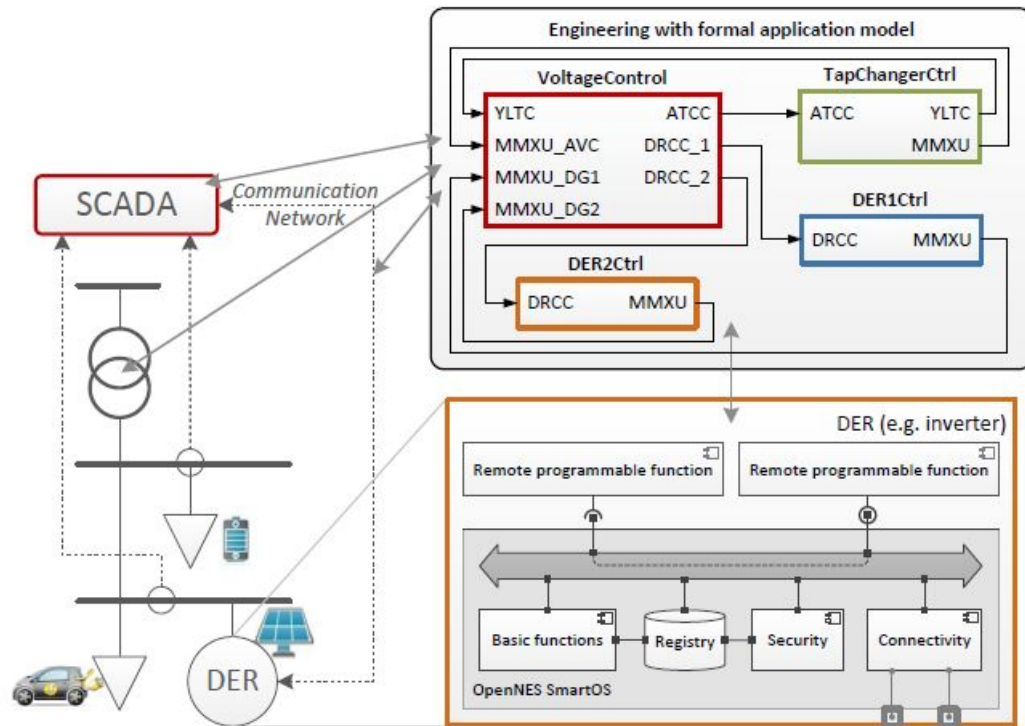


Abbildung 2.1: Übersicht des OpenNES Konzepts [?]

2.2 Basisfunktionen

Wie in Abbildung 2.1 zu sehen ist, beinhaltet das Basis Funktionsmodul Standardmethoden für **DER!**/Inverter Systeme [?]. Dieses interne Modul bietet Funktionen zur Unterstützung der Stromnetzwerke. Aktuell dienen die Funktionen in erster Linie dazu, die Einspeiseleistung zu maximieren, indem die negativen Auswirkungen auf die Netzparameter (Spannung, Frequenz) möglichst minimiert werden und verhindert wird, dass das Netz destabilisiert wird. OpenNES bietet Netzbetreibern eine gezielte, aktive Unterstützung, um Systeme und den Stromfluss großflächig zu beeinflussen. Es können weitere DERs (z.B.: Photovoltaik Anlagen) ohne den Ausbau der Netze (engl. Grid Enforcement) installiert werden. Diese Funktionen und Verhaltensmuster werden mit der Bezeichnung Advanced Grid Features (AGF) gekennzeichnet und dienen dazu, Anforderungen an das System zu bewältigen.

2.3 Softwarekomponenten

Die **SWK!** sind als eigenständiger Bestandteil in das SmartOS eingebunden. Sie sind austauschbar, das heißt man kann die Komponenten dynamisch hinzufügen oder ent-

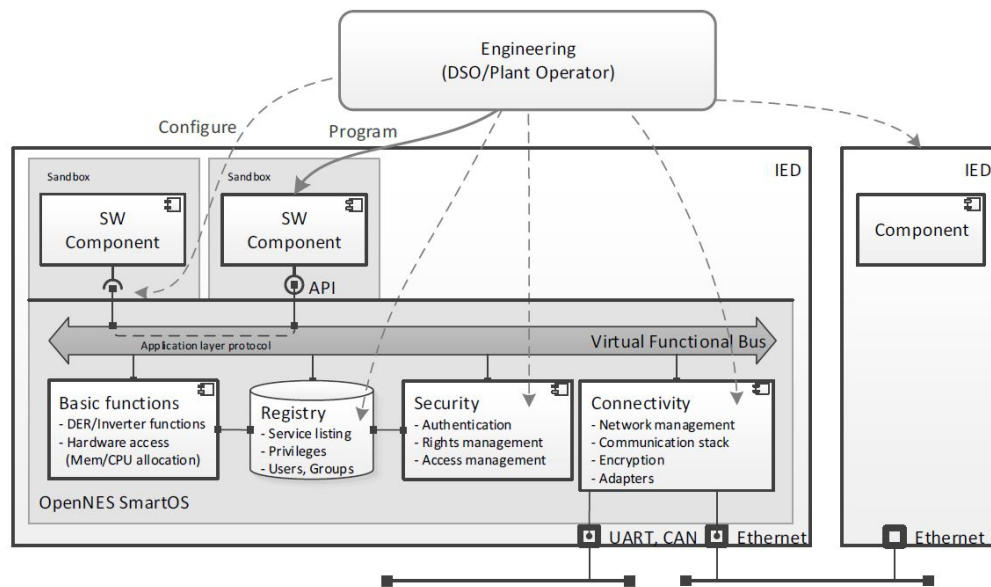


Abbildung 2.2: OpenNES System Architektur [?]

fernen, ohne dass man das System neu kompilieren muss. Diese können entweder von den DER Herstellern oder von einem zertifizierten externen Anbieter entwickelt und geliefert werden und in verschiedenen Sprachen bzw. mit Hilfe verschiedener Technologien implementiert werden.

Wichtig für die Sicherheit ist, dass die **SWK!** in einer sicheren Umgebung eingekapselt werden wie in Abbildung 2.3 zu sehen ist. Dies wird durch die Sandbox Methode erreicht, d.h. dass sie von den Systemressourcen isoliert werden. Sie werden in einer Laufzeitumgebung z.B.: Java Runtime Environment ausgeführt. Dies ermöglicht eine weitere Verkapselung aus dem System. Dadurch haben die **SWK!** einen eingeschränkten Zugang z.B. auf Speicher- und Dateiverwaltung. Jede Komponente kann über den **VFB!** (**VFB!**) Dienste angeben, bereitstellen und anfordern. Das Updaten und Programmieren der Komponenten kann während der Laufzeit erfolgen. Dies bringt einige Vorteile wie z.B.: neue Funktionalitäten oder das Rekonfigurieren der Komponenten, aber auch Gefahren mit sich. Es sollte sichergestellt werden, dass die Anwendung nur innerhalb bestimmter Einschränkungen arbeitet, z.B.: das Ändern der Blindleistung der **DER!** nur zwischen $\pm 10\%$ der Nennleistung.

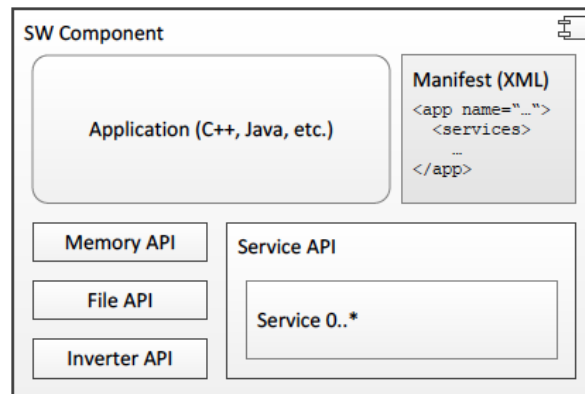


Abbildung 2.3: Softwarekomponente [?]

2.4 Sicherheitsmodul

Jegliche Art von Kommunikation zwischen den Komponenten, auch jene zwischen den Softwarekomponenten des gleichen Geräts, müssen durch das Sicherheitsmodul autorisiert werden. Die Sicherheit wird über die rollenbasierte Zugriffskontrolle (RBAC) geregelt und dient als Zugriffspunkt. Für jede Zugriffsanforderung werden die Zugriffsrechte durch das Modul bestimmt. Der **VFB!** verarbeitet nur die Daten, die vom Sicherheitsmodul genehmigt wurden [? ?].

2.5 Virtual Functional Bus

Der **VFB!** ermöglicht die Kommunikation zwischen den einzelnen Komponenten. Alle Software Komponenten können über eine API Schnittstelle Dienste anbieten oder beziehen. Die API wird vom OpenNES Smart OS über den VFB zur Verfügung gestellt. Die Dienste sind rein intern oder extern zugänglich [?].

2.6 Registrierung

Das Registrierungsmodul beinhaltet alle Benutzer oder Gruppen, die dem **IED!** bekannt sind, sowie die Berechtigungen und die Daten der Software Komponenten [?]. Dieses Modul ist der Berechtigungsspeicher des OpenNES Systems.

2.7 Connectivity Modul

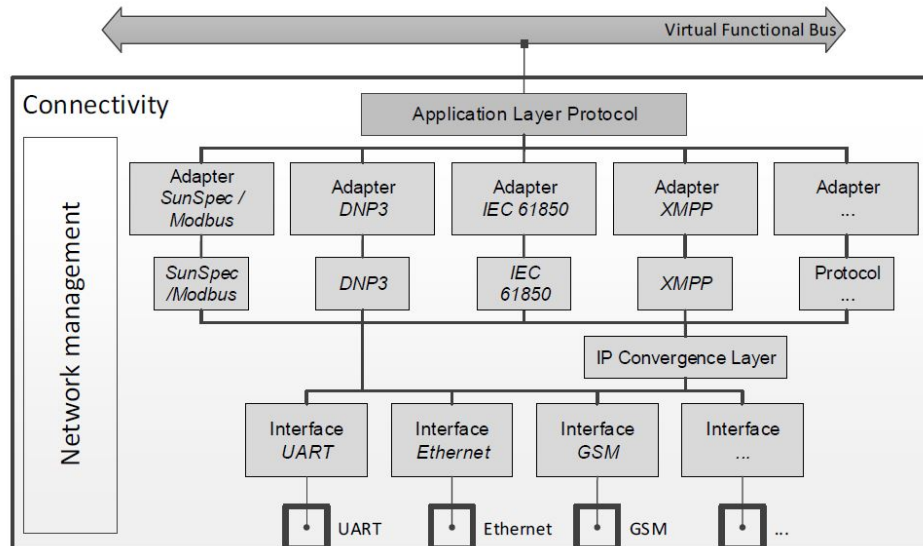


Abbildung 2.4: Connectivity Modul [?]

Das **CM!** (**CM!**) (Abbildung 2.4) beinhaltet die Schnittstellen zur externen Kommunikation über den virtuellen Bus welcher als **VFB!** bezeichnet wird [?]. Diese Projektarbeit konzentriert sich hauptsächlich, auf dieses Modul. Durch die Verwendung von Protokolladaptern werden die Flexibilität und die Erweiterungsmöglichkeiten des Systems erhöht. Protokolladapter werden verwendet, um zwischen den verschiedenen Verbindungsmethoden und den einheitlichen Protokollen der Anwendungsschicht zu übersetzen. Auch werden die Adapter dazu verwendet, fehlende Funktionen und Kommunikationsschema wie z.B.: publish-subscribe, request-response, push messaging pattern u.a. anbieten zu können.

3 Basistechnologien

3.1 Modbus Protokoll

In diesem Kapitel wird das Modbus Protokoll beschrieben. Der Protokolladapter ist für Modbus TCP entwickelt worden, deshalb wird auch vorwiegend diese Version erklärt. Entwickelt wurde das Protokoll aber ursprünglich für die serielle Schnittstelle. Daher wird zuerst das Protokoll für die serielle und anschließend die Erweiterung für TCP beschreiben.

3.1.1 Geschichte

Das Modbus Protokoll für die serielle Schnittstelle wurde von der Firma Modicon entwickelt und 1979 veröffentlicht. Der Name der Firma Modicon ist heute Schneider Electric. Mit Modbus TCP/IP wurde von Schneider Electric 1999 ein offener Standard entwickelt, um Modbus auch über Ethernet nutzbar zu machen. Die Rechte an Modbus wurden 2004 an die Modbus Organization übergeben, welche zurzeit das Protokoll betreut [?].

3.1.2 Grundlagen

Modbus ist ein Protokoll, welches sich auf den Schichten 2 und 7 des OSI Schichtmodells befindet. Das Protokoll stellt eine Client-Server Verbindung zwischen den Teilnehmern her. Es funktioniert nach dem Request-Response Prinzip. Die Funktionen des Protokolls werden durch sogenannte Funktionscodes spezifiziert [?].

Abbildung 3.1 stellt die unterschiedlichen Modbus Varianten dar.

- Modbus TCP/IP
Master/Slave Kommunikation über Ethernet
- Modbus RTU
serielle Master/Slave Kommunikation über RS232 oder RS485
- Modbus Plus
ein Netzwerk mit Tokenweitergabe

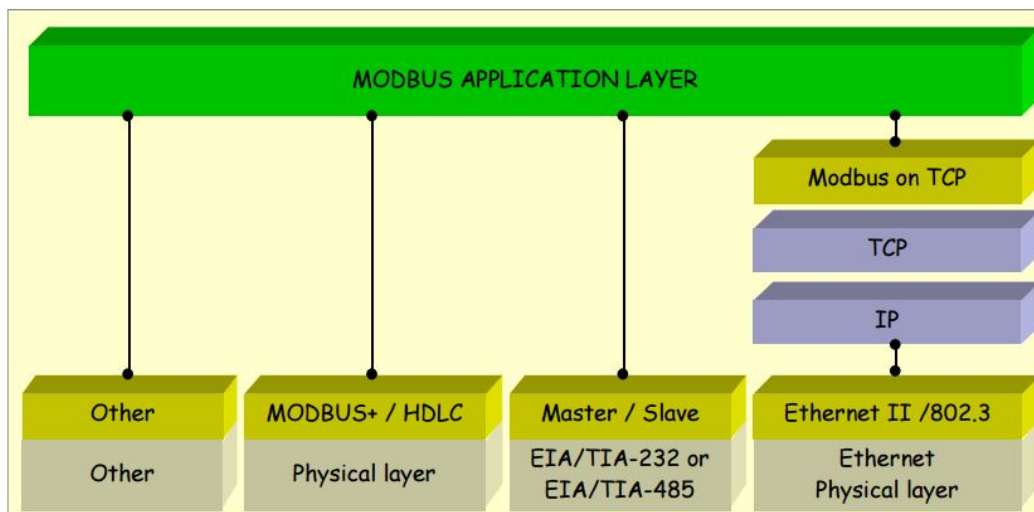


Abbildung 3.1: Schichten des ModBus Protokolls [? , S.2]

3.1.3 Protokollaufbau

Wie aus Abbildung 3.2 ersichtlich, besteht die **PDU!** (**PDU!**) aus Funktionscode und den Daten. Für den Funktionscode ist ein Byte reserviert. Daraus ergeben sich 256 Funktionscodes, wobei der Bereich 128 - 255 reserviert ist für die Exception Antworten und 0 ein ungültiger Funktionscode ist. Für die Funktionscodes ergibt sich daraus ein nutzbarer Bereich von 1 - 127. Das Modbus Protokoll ist ursprünglich für die serielle Schnittstelle entwickelt worden. Dabei wurde die Application Data Unit (ADU) auf 256 Bytes begrenzt. Von den 256 Bytes werden 1 Byte für die Modbus Server Adresse verwendet und 2 Bytes für die **CRC!** (**CRC!**). Daraus ergibt sich eine Länge von 253 Bytes für die PDU. Weiters lässt sich dadurch für das Datenfeld eine Länge von 252 Bytes berechnen. Modbus verwendet die Big-Endian Byte Reihenfolge für Daten und Adressen [?].

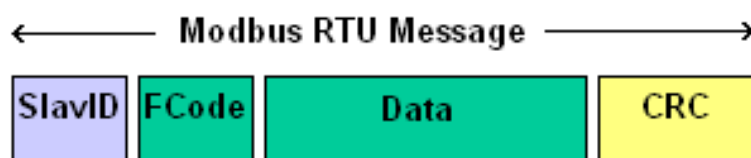


Abbildung 3.2: Schematische Darstellung eines Modbus RTU Frames [?]

Es werden 3 PDUS von Modbus spezifiziert

- Modbus request PDU
Diese wird vom Client generiert und an den Server übermittelt.
- Modbus response PDU
Diese wird vom Server als Antwort an den Client gesendet, wenn die Aktion am Server erfolgreich war.
- Modbus exception response PDU
Diese wird vom Server als Antwort an den Client gesendet, wenn am Server während der Aktion ein Fehler auftrat.

Abbildung 3.3 zeigt den Ablauf einer erfolgreichen Modbus Transaktion. Dabei wird durch den Funktionscode und den dazugehörigen Daten ein Request an den Server gesendet. Anschließend wird die PDU mit demselben Funktionscode und den Nutzdaten wieder an den Client gesendet [?].

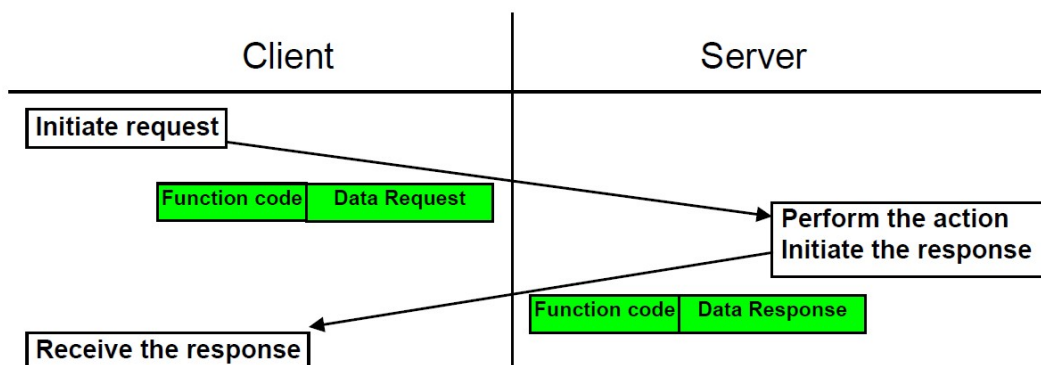


Abbildung 3.3: Fehlerfreie Modbus Transaktion [? , S.4]

Abbildung 3.4 zeigt den Ablauf einer fehlerhaften Modbus Transaktion. Dabei wird wie zuvor schon beschrieben ein Request an den Server gesendet. Am Server wird die Anfrage überprüft und als ungültig erkannt. Nun wird für die Antwort der Funktionscode der Anfrage kopiert und das höchste bit gesetzt. Der daraus resultierende Code wird Errorcode genannt. In das Datenfeld der Response PDU wird dann der Errorcode in das Feld des Funktionscode kopiert und in das darauf folgende Feld wird ein Exceptioncode geschrieben. Diese PDU wird dann wieder an den Client gesendet [?].

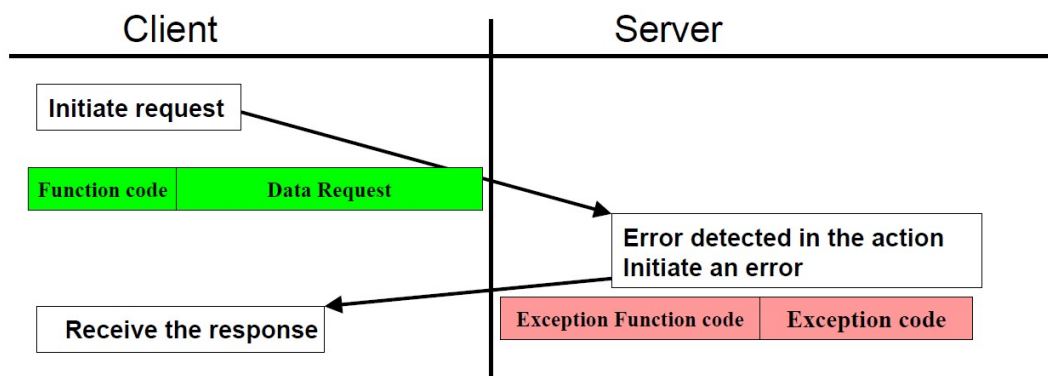


Abbildung 3.4: Exception Antwort einer Modbus Transaktion [? , S.4]

3.1.4 Modbus Datenmodell

In Tabelle 3.1 werden die 4 unterschiedlichen Objekttypen, die von Modbus unterstützt werden, dargestellt. Daraus ist ersichtlich, dass eine Adresse auf nur 1 Bit oder auf einen 16 bit langen Bereich verweisen kann. Für die Adresse wird ein Wort (2 Byte) verwendet. Das bedeutet, es können von jedem Objekttyp 65536 Einträge am Modbus Server adressiert werden. Die Modbusadresse verweist auf den Wert, der von der Applikation, auf der der Server läuft, im Speicher gehalten wird. Abhängig vom Objekttyp kann der Wert von der Applikation bzw. vom Modbus Server gelesen oder gelesen/beschrieben werden. Daraus ergeben sich unterschiedliche Möglichkeiten, das Modbus Datenmodell aufzubauen. Es kann ein Datenmodell aufgebaut werden, in dem sich alle Objekttypen einen Bereich teilen. Die zweite Möglichkeit ist, dass ein Modell aufgebaut wird, bei dem jeder Objekttyp auf einen separaten Datenbereich zeigt. Welche Daten hinter den Registeradressen stehen, ist der Applikation, welche auf dem Modbusserver läuft, überlassen [? ?].

Register	Länge	Funktion	Beschreibung
Discrete Input	1 bit	read	Digitaleingang, welcher vom Client gelesen wird
Coil	1 bit	read/write	Digitalausgang, welcher vom Client beschrieben wird
Input Register	16 bit	read	Bereich für die Darstellung eines Werts für den Client
Holding Register	16 bit	read/write	Bereich für das Schreiben eines Werts an den Server

Tabelle 3.1: Modbus Datenmodell [?]

Abbildung 3.5 zeigt schematisch den Aufbau eines Modbus Geräts. Links ist die Applikation (*Device application*), welche am Gerät läuft, dargestellt. Die grauen Felder sollen den Speicher der Anwendung darstellen. Abhängig vom Objekttyp werden die Werte im Speicher von der Applikation und vom Modbus Server gelesen und geschrieben. In der Mitte des Bildes ist das Modbus Datenmodell mit den Registeradressen dargestellt, welche auf den gemeinsamen Speicher zeigen. Rechts sind die Modbus Transaktionen dargestellt, die Daten aus dem Gerät, über die Register, lesen bzw. schreiben können [?].

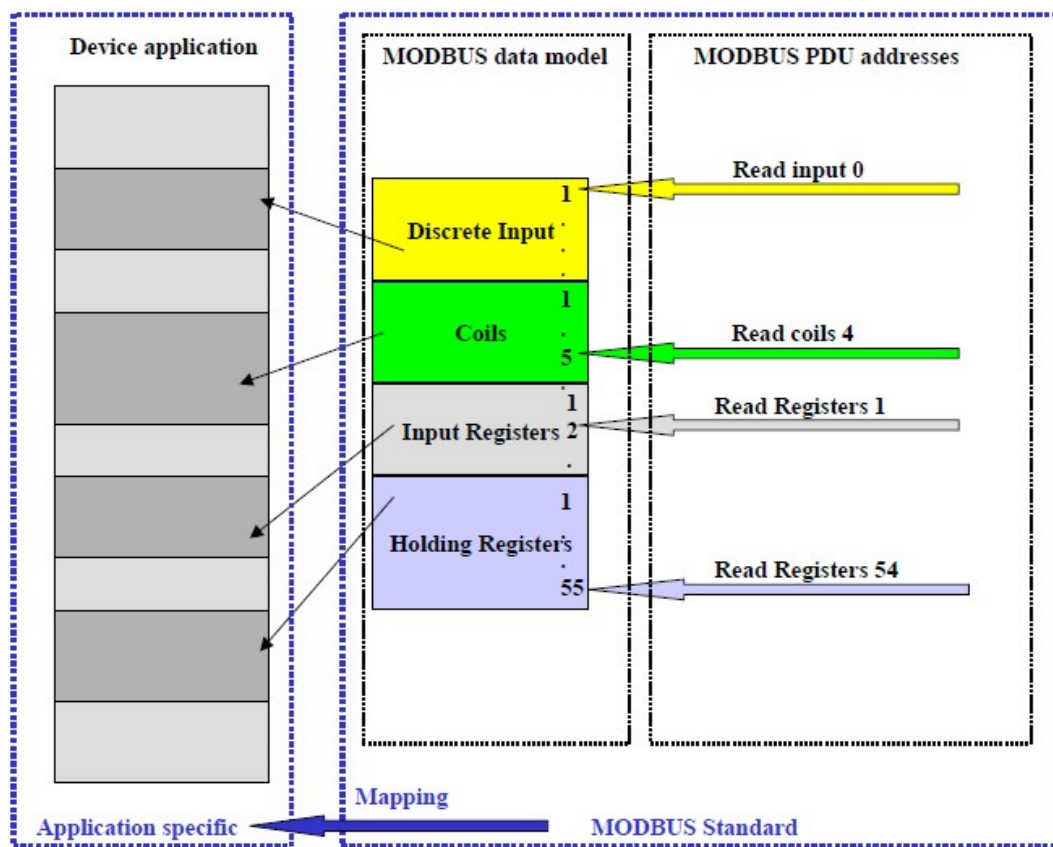


Abbildung 3.5: Modbus Adressmodell [? , S.8]

3.1.5 Funktionscodes

In diesem Abschnitt werden die zwei wichtigsten Funktionscodes, welche für die Implementierung benötigt werden beschrieben. Für diese werden nur Holding Register benötigt. Deshalb werden hier nur die Funktionscodes behandelt, mit denen Holding Register beschrieben oder gelesen werden können [?].

Lese Holding Registers

Bei der Verwendung des Funktionscode 0x03 wird ein zusammenhängender Block an *Holding Register* gelesen. Wie Tabelle 3.2 zeigt, müssen in der PDU die Startadresse und die Anzahl an zu lesenden Registern übergeben werden. Ist die Transaktion erfolgreich am Server, dann wird derselbe Funktionscode zurückgesendet. In der PDU wird im ersten Byte die Anzahl an Werten zurückgegeben. Im übrigen Datenbereich befinden sich die Werte, auf welche die Register des Servers zeigen. Dies ist allgemein in Tabelle 3.3 dargestellt. Tabelle 3.4 zeigt die Antwort bei einem Fehler der Transaktion am Server. Tritt ein solcher Fehler auf, dann wird zum Funktionscode 0x80 addiert. Dieser neue Funktionscode wird zurück an den Client gesendet. Weiters wird in das Datenfeld der Exceptioncode geschrieben. Dieser gibt Aufschluss über die Art des Fehlers und kann vom Client ausgewertet werden [?].

Funktionscode	1 Byte	0x03
Startadresse	2 Byte	0x0000 ... 0xFFFF
Anzahl der Register	2 Byte	1 ... 125

Tabelle 3.2: Anforderung Lese Holding Registers [?]

Funktionscode	1 Byte	0x03
Anzahl an Bytes	1 Byte	2 x N*
Register Wert	N* x 2 Byte	

Tabelle 3.3: Antwort Lese Holding Registers wenn erfolgreich [?]

Fehler code	1 Byte	0x83
Exception code	1 Byte	01, 02, 03, 04

Tabelle 3.4: Antwort Lese Holding Registers wenn Fehler [?]

Schreibe Multiple Registers

Bei der Verwendung des Funktionscodes 0x10 können mehrere Werte in einem zusammenhängenden Block von Holding Registern geschrieben werden. Byte 0 der PDU ist der Funktionscode. In der PDU müssen das erste und zweite Byte die Registeradresse sein. Byte drei und vier sind die Anzahl der Register, welche beschrieben werden sollen. Byte fünf ist die Anzahl an zu schreibenden Bytes. Abhängig davon wie viele Werte geschrieben werden müssen, ist die restliche Anzahl an nicht verwendeten Bytes in der PDU für die Werte die geschrieben werden sollen vorgesehen. Dies zeigt Tabelle 3.5. Tabelle 3.6 stellt den generellen Aufbau der PDU dar, wenn das Schreiben der Register erfolgreich war. Hier wird wieder der selbe Funktionscode vom Server zurückgesendet. Weiters steht in den Daten die Offset Registeradresse, ab der die Daten an den Server geschrieben wurden, sowie die Anzahl an Registern. Bei einem Fehler wird der Aufbau der Antwort vom Servers gleich, wie schon in Abschnitt 3.1.5 generiert. Spezifisch für die Funktion Schreibe Multiple Registers ist dies in Tabelle 3.7 dargestellt [?].

Funktionscode	1 Byte	0x10
Startadresse	2 Byte	0x0000 ... 0xFFFF
Anzahl der Register	2 Byte	1 ... 125
Anzahl an Bytes	1 Byte	2 x N*
Register Werte	N* x 2 Byte	

Tabelle 3.5: Anforderung Schreibe Multiple Registers [?]

Funktionscode	1 Byte	0x10
Startadresse	2 Byte	0x0000 ... 0xFFFF
Anzahl der Register	2 Byte	0x0000 ... 0xFFFF

Tabelle 3.6: Antwort Schreibe Multiple Registers wenn erfolgreich [?]

Fehler code	1 Byte	0x90
Exception code	1 Byte	01, 02, 03, 04

Tabelle 3.7: Antwort Schreibe Multiple Registers wenn Fehler [?]

3.1.6 ModBus TCP/IP

Bei Modbus TCP/IP oder kurz Modbus TCP sind Server und Client über ein Ethernet TCP/IP Netzwerk miteinander verbunden. Die PDU des Modbus Protokoll wurde bereits im Abschnitt 3.1.3 beschrieben. Dieses Kapitel behandelt die ADU bei der Verwendung des TCP. Anhand von Abbildung 3.2 ist bereits der Aufbau eines Modbus RTU Frames erklärt worden. Bei diesem besteht die ADU aus *Slave ID* und **CRC!**. Diese werden bei einem seriellen Netzwerk verglichen mit dem OSI Schichtmodell an der untersten Schicht hinzugefügt bzw. implementiert. Bei der Verwendung von TCP kommt zur Modbus PDU der **MBAP!** (**MBAP!**) Header hinzu. In diesem ist nun die *Slave ID* welche bei TCP *Unit ID* genannt wird, enthalten. Bild 3.6 stellt auch die weiteren Felder des MBAP Headers dar. Diese bestehen aus der *Transaction ID*, welche vom Client bei einer Transaktion vergeben wird, weiters die *Protocol ID*, welche für Modbus immer 0 ist und das Feld *Length*, das die Anzahl der darauffolgenden Bytes angibt. CRC wird nicht in die TCP ADU übernommen. Diese Funktion erledigen die darunterliegenden Protokolle [? ? ?].

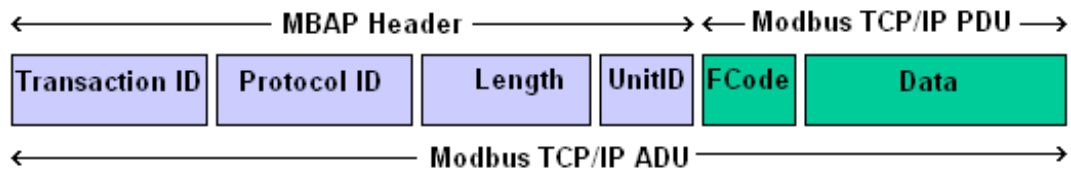


Abbildung 3.6: Modbus ADU Vergleich [?]

Abbildung 3.7 zeigt prinzipiell, wie die Modbus TCP ADU in den TCP Frame eingepackt wird. Auch wird dargestellt, dass Ethernet nun den *Error Check* durchführt. Für Modbus TCP ist der Port 502 reserviert [?].

CONSTRUCTION OF A TCP/IP-ETHERNET DATA PACKET

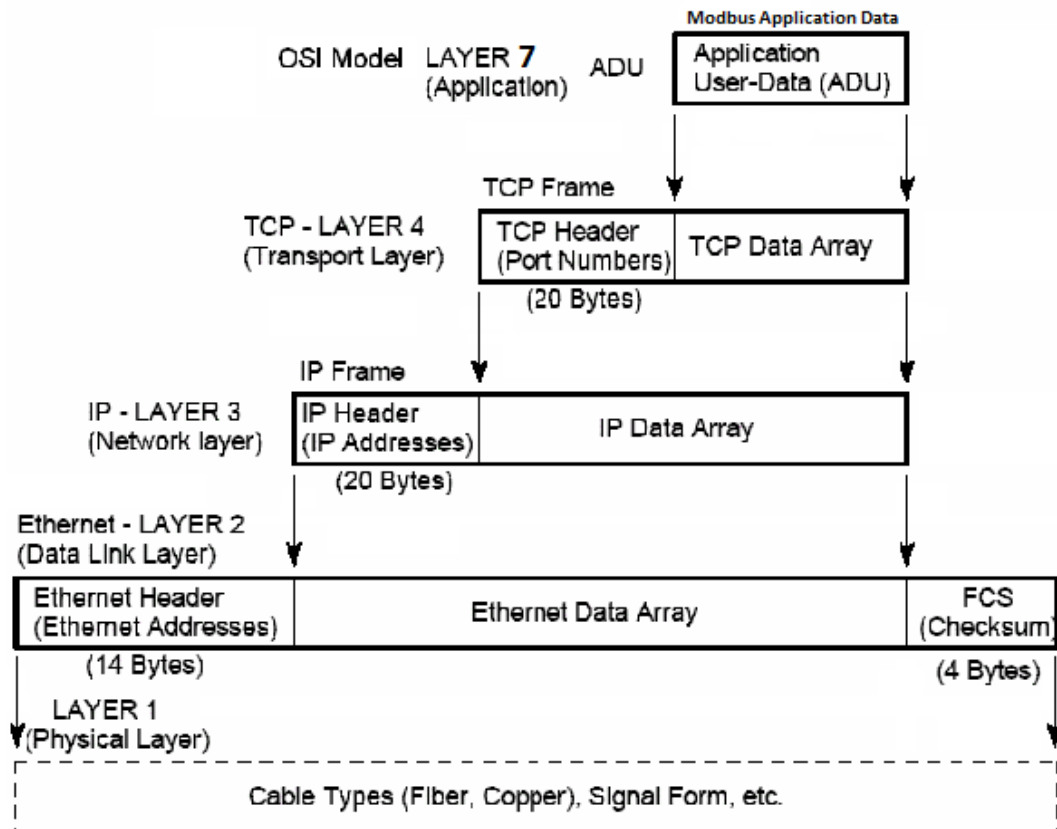


Abbildung 3.7: Modbus Encapsulation bei TCP/IP [?]

Wie schon in Abschnitt 3.1.3 beschrieben, ist für die Slave ID nur ein Byte reserviert. In einem seriellen Netzwerk können so theoretisch 256 Server adressiert werden. 0 ist die Broadcast Adresse. Die Adressen 248 - 255 sind reserviert. Daraus ergeben sich gültige Adressen von 1 - 247 für die Server in einem seriellen Netzwerk. Ein Modbus TCP Gerät kann somit auch nur auf 247 Server routen. Abbildung 3.8 zeigt, wie ein TCP Gerät als Gateway zu einem seriellen Netzwerk fungiert. In TCP/IP Netzwerken ist jedoch die Anzahl an Modbus Servern, die adressiert werden können, nicht mehr von der Slave ID alleine abhängig. Jedes Modbus Gerät in diesem Netzwerk hat eine IP Adresse und somit ist die Anzahl abhängig von den möglichen IP-Adressen in einem Netzwerk [? ?].

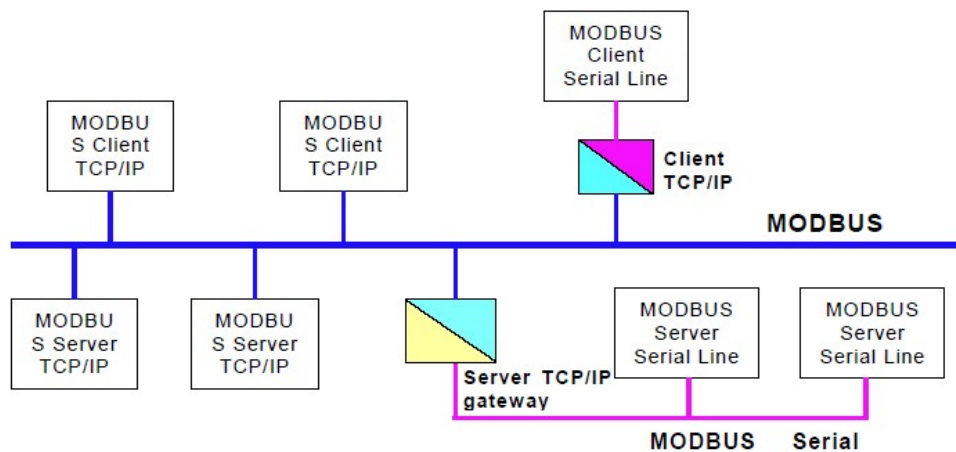


Abbildung 3.8: Modbus TCP/IP Kommunikations-Übersicht [? , S.4]

3.2 SunSpec

3.2.1 Übersicht

Die SunSpec Alliance hat es sich zur Aufgabe gemacht, einen de facto Standard zu entwickeln, um für erneuerbare Energiequellen eine einheitliche Schnittstelle zu schaffen. Ziel ist es dabei durch, den einheitlichen Standard die Kosten für die Errichtung neuer Anlagen oder auch die Implementierung neuer Geräte in ein bestehendes System zu senken. Unabhängig vom Hersteller des Modells soll auf Geräte wie Wechselrichter, Stromzähler, Panels usw. einheitlich zugegriffen werden können [?].

3.2.2 SunSpec Informationsmodell

Aufgabe des Informationsmodells ist es, die Datenpunkte eines Geräts auf das verwendete Kommunikationsprotokoll zu abbilden. Bei der Verwendung von Modbus TCP werden die Datenpunkte eines Gerätes eindeutig auf Registeradressen des Modbus Protokolls abgebildet. Hat der Client dasselbe Informationsmodell hinterlegt, kann dieser die Daten, welche in den Registern bereitgestellt werden wieder auf für die Applikation sinnvolle Daten zurückführen. Für ein gültiges SunSpec Gerät müssen Informationsmodelle zu einer Gerätebeschreibung strukturiert werden.

Es gibt 4 Typen von Informationsmodellen. Diese heißen folgendermaßen:

- Common Model
- Standard Model(s)
- Vendor Model(s)
- End Model

Es werden mindestens 3 Informationsmodelle für eine Gerätebeschreibung benötigt. Dazu gehören das Common Model am Beginn und das End Model am Ende der Beschreibung, dazwischen muss sich mindestens ein Standard Model oder Vendor Model befinden. Das erste Register eines Informationsmodells verweist immer auf eine eindeutige ID. Im zweiten Register befindet sich die Länge des Modells. Das End Model markiert das Ende des SunSpec Gerätes. Dadurch kann ein Client durch alle Register iterieren und abhängig von den für ihn gültigen IDs die Daten nutzen [?].

3.2.3 SunSpec Datentypen

Modbus stellt wie bereits beschrieben nur 1 bit oder 16 bit lange Datentypen zur Verfügung. Für die Datenpunkte der SunSpec Implementierung werden aber Datentypen benötigt die größer als 2 Byte sind. Dieses Kapitel gibt eine Übersicht der verwendeten Datentypen. Diese sind wie folgt:

- int: Integer Vorzeichenbehaftet
- uint: Integer Vorzeichenlos
- pad: Padding Feld, um ein Informationsmodell mit Leerdaten aufzufüllen
- acc: wird für Daten verwendet die akkumuliert werden wie z.B. Energieverbrauch.
- enum: Aufzählungstyp, wird verwendet für Status
- bitfield: Eine Zusammenfassung von bits.
- string: Zeichenkette
- ip: Datentyp für eine IP-Adresse

Ganzzahlige Datentypen verwenden gleich wie Modbus die Big-Endian Byte Reihenfolge.

Eine Aufzählung der Datentypen und deren Speicherbelegung.

- Integer
Ein 16 bit Integer benötigt genau ein Modbus Register. Dies soll Tabelle 3.8 zeigen. Weiters soll diese Tabelle als Beispiel für größere Ganzzahlen wie 32, 64 und 128 bit Integer dienen. Der Aufbau ist bei diese Datentypen derselbe, nur dass dementsprechend mehr zusammenhängende Register benötigt werden.

MB Register	1															
Byte	0								1							
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Tabelle 3.8: 16 bit Integer [?, S.14]

- String

String plus die Längenangabe definiert die Anzahl an Zeichen. Tabelle 3.9 soll dies am Beispiel von einem SunSpec *string16* zeigen. Die Registeranzahl für einen String wird folgendermaßen berechnet. Zuerst wird die Anzahl der Zeichen des Strings durch zwei dividiert. Das Ergebnis dieser Division aufgerundet auf die nächsthöhere Ganzzahl ergibt die Anzahl der benötigten Register. Mit NULL (ASCII 0x0) kann ein String frühzeitig abgeschlossen werden.

MB Register	1		2		3		4		5		6		7		8	
Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Character	E	X	A	M	P	L	E	spc	S	T	R	I	N	G	!	NULL

Tabelle 3.9: String [?, S.15]

- Gleitkommadarstellung

Eine Gleitkommazahl wird 32 bit codiert nach IEEE 754. Tabelle 3.10 zeigt, welche Registerbits für welche Felder der Gleitkommazahl verwendet werden.

MB Register	1			2		
Byte	0		1	2	3	
Bits	31	30 ... 24	23 ... 16	15 ... 8	7 ... 0	
IEEE 754	Vorzeichen	Exponent	Mantisse			

Tabelle 3.10: Float [?, S.16]

- Skalierungsfaktoren

Wenn keine Gleitkommazahlen in der Applikation des Clients verwendet werden können, ist es möglich, die Datenpunkte als Integer Werte darzustellen. Der Skalierungsfaktor schiebt dabei das Komma um n Stellen nach links oder rechts. Der Wertebereich von n umfasst -10 bis 10. Das am Server verwendete Informationsmodell definiert, ob Gleitkommazahlen oder Ganzzahlen mit Skalierungsfaktoren verwendet werden [?].

3.2.4 Register Mapping

Bei der Verwendung von Modbus und SunSpec befindet sich in der Basis Registeradresse die 32bit lange SunS ID mit dem Wert 0x53756e53. Das Basis Register muss immer diesen Wert zurückgeben. Wird dieser Wert nicht zurückgegeben, dann muss dieser in einem der alternativen Basis Register stehen. Ist dies nicht der Fall, dann handelt es sich nicht um ein SunSpec Gerät. Nach den 2 Registern für die SunS ID muss im folgenden Register die ID für das Common Model stehen. Der restliche Teil der Register ist so aufgebaut, wie im Unterkapitel 3.2.2 beschrieben [?].

SunSpec gibt folgende Basis Register vor:

- Basis Register: 40001
- Alternatives Register: 50001
- Alternatives Register: 00001

4 Entwurf

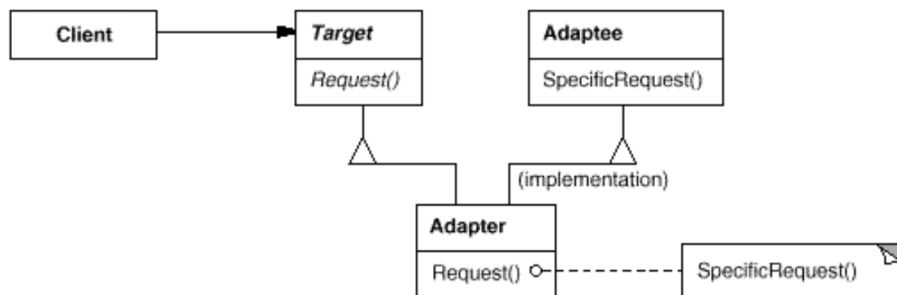
In diesem Kapitel wird das Adaptermuster im Allgemeinen und die Anforderungen bzw. die Umsetzung des **PA!**s betrachtet. Wie in Kapitel 2.7 beschrieben, erhöhen die jeweiligen Protokolladapter die Flexibilität und die Erweiterungsmöglichkeiten des OpenNES Systems.

4.1 Übersicht Adapter Pattern

In [?] wird ein Adapter wie folgt definiert:

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces"([?] Seite 157).

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:

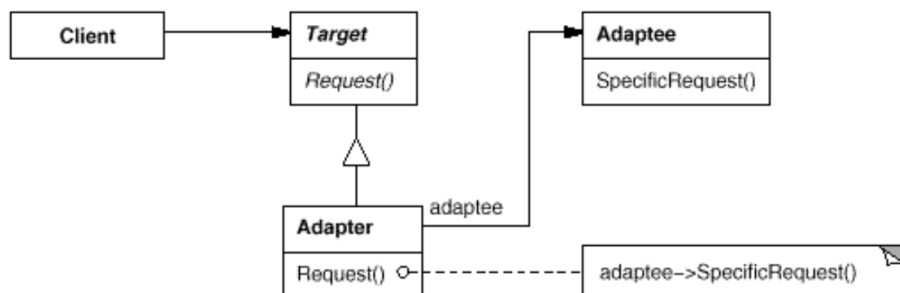


Abbildung 4.1: UML Diagramm des Objekt- bzw. Klassenadapters [?, S.159]

Wie in Abbildung 4.1 dargestellt, gibt es zwei Möglichkeiten, die Adapter zu implementieren. Klassen- und Objektadapter verwenden verschiedene Arten der Adaption. Der Klassenadapter bildet eine Unterklasse des Adaptee und des Ziels, wohingegen der Objektadapter die Zielschnittstelle implementiert [? ?].

Die Adapter Pattern werden wie in [?] beschrieben verwendet:

- wenn das Interface der bestehenden Klasse nicht zur benötigten Klasse passt
- wenn wiederverwendbare Klassen benötigt werden, die über keine kompatiblen Schnittstellen verfügen
- um Unterklassen in ein System zu implementieren, ohne die Schnittstellen der Unterklassen zu verändern

Der OpenNES Protokolladapter entspricht vor allem dem letzten Punkt. Wie in der Abbildung 2.7 zu sehen ist, verhält sich der Adapter als Unterklasse der jeweiligen Protokolle. Dadurch können die Funktionscodes des Modbus Protokolls so genutzt werden, dass das geforderte Messaging Muster dabei erfüllt wird.

4.2 Anforderungsanalyse

Im Zuge der Bachelorarbeit werden folgende Anforderungen an den **PA!** (**PA!**) gestellt. Der **PA!** im OpenNES SmartOS soll Teil des übergeordneten Connectivity Moduls sein. Da diverse Netzwerkstrukturen bereits vorhanden sind und unterschiedliche Protokolle wie IEC 61850, ModBus, XMPP, DNP3 etc. verwendet werden, soll der **PA!** ein Interface zu **IED!**s (**IED!**s) und **SED!**s schaffen. Als **IED!**s werden in diesem Kontext OpenNES fähige Geräte bezeichnet und als **SED!**s die Legacy Geräte. Der Adapter soll dabei die Integration der Legacy Geräte ins OpenNES ermöglichen. Er soll eine Schnittstelle zur unteren Modbusschicht bieten und folgende Funktionen beinhalten:

- *Push 1:1*
Nachricht wird vom Sender zum Empfänger übertragen (siehe Kapitel 5.9).
- *Push 1:N*
Bei der Push 1:n Funktion schreibt ein Client den Wert eines Datenpunktes auf eine Anzahl von n Server (siehe Kapitel 5.10).

- *Request-Response*

Request Response besteht aus 2 Teilnehmern, Sender und Empfänger. Diese Funktion fragt bestimmte Datenpunkte in Registern ab (z.B.: Holding Register) (siehe Kapitel 5.10).

- *Publish-Subscribe*

Publish-Subscribe ist auch als Observer Pattern bekannt. Bei diesem Pattern sendet der Protokolladapter (Client) Anfragen an den Modbus Server und überprüft dabei, ob sich im Server Register Daten verändert wurden (siehe Kapitel 5.11).

Weiters soll das Adressmapping zwischen der OpenNES Adressierung und der Adressierung der Adapter wie in Abbildung 4.2 umgesetzt werden.

```
AddressMapping =
OpenNESAddress ":" "{" AdapterName "[" AdapterAddress "]" ("," AdapterName "[" AdapterAddress "]"*)* "}"|
```

Abbildung 4.2: Adressmapping OpenNES zu Adapter

Es bestehen mehrere Protokollstandards, wie in Punkt 4.2 beschrieben wurde. Für dieses Projekt soll das *ModbusTCP/Sunspec Protokoll* verwendet werden.

Da der ModbusTCP keine Datenpunkte abbildet, wird der SunSpec Standard verwendet. Weiters soll der Adapter die vom OpenNES an das Connectivity Modul übergebene Adresse des Servers in die Remote-Adresse für den Modbus übersetzen. Das Ziel ist es, dass ein **IED!** mit mehreren **SED!**s kommuniziert. Die Befehle, von welchem **SED!**s Informationen benötigt werden bzw. Daten geschrieben werden, kommen seitens des **CM!**.

5 Implementierung

5.1 Umsetzung

In diesem Kapitel wird die Umsetzung der Funktionen (Messaging Muster) des Protokolladapters beschrieben. Die Funktionscodes seitens Modbus funktionieren wie bereits in Kapitel 3.1.5 beschrieben nach dem Request-Response Prinzip. Es wird ein Funktionscode an den Server gesendet und dieser oder ein Exceptioncode wird vom Client wieder empfangen. Die Funktionen des Protokolladapters befinden sich eine Ebene darüber. D.h. die Funktionscodes des Modbus Protokolls werden so genutzt dass das geforderte Messaging Muster dabei erfüllt wird. Ebenfalls werden in diesem Kapitel die einzelnen Klassen, die erstellt wurden, beschrieben.

5.2 Ausgewählte Programmiersprache und IDE

Die Entwicklung des Protokolladapters erfolgte in der Programmiersprache Java. Als Entwicklungsumgebung wurde Eclipse Jee Neon (4.6.2) verwendet. Um das Modbus Protokoll zu implementieren wurde die Library Modbus4j¹ gewählt. Dazu wurde ein neues Maven Projekt angelegt. Als Laufzeitumgebung ist die Version jre1.8.0 verwendet worden.

5.3 SunSpecDataType

Der Enum Typ *SunSpecDataType* ist notwendig um den Wert, welcher von einem Modbus Server gelesen wird, in einen Java Datentyp bzw. Objekt umzuwandeln. Das Listing 5.1 zeigt die möglichen Datentypen.

```
1 public enum SunSpecDataType {  
2     SHORT,  
3     INT,  
4     LONG,  
5     FLOAT,  
6     STRING  
7 }
```

Listing 5.1: SunSpecDataType

¹<https://github.com/infiniteautomation/modbus4j>

5.4 Status

Der Enum Typ “Status“ listet die möglichen Meldungen auf, welche eine Funktion des Adapters zurückgeben kann. Da zwischen den Modulen und den Connectivity Modul ein Datenaustausch stattfindet, der unabhängig vom verwendeten Protokolladapter ist, wird der Status generisch gehalten. Es ist nicht notwendig, dabei die einzelnen spezifischen Exceptions auszugeben. Prinzipiell ist es wichtig zu wissen, ob die Transaktion erfolgreich war. War diese nicht erfolgreich, benötigen die Module, welche über das Connectivity Modul senden und empfangen keine Modbus Exception, da diese nicht wissen über welchen Protokolladapter kommuniziert wird. Listing 5.2 gibt einen Überblick der Status.

```
1 public enum Status {  
2     Nothing,  
3     Error,  
4     WertGeschrieben,  
5     WertEmpfangen,  
6     KeineVerbindungsparameter,  
7     KeinGueltigerDatentyp  
8 }
```

Listing 5.2: Status

5.5 Remoteaddress

Wie bereits beschrieben, empfängt das Connectivity Module eine generische Adresse. Diese wird vom Connectivity Module in die Remoteaddress für Modbus umgesetzt. Tabelle 5.1 zeigt die Auflistung der Parameter, aus denen die Remoteaddress besteht. Die Remoteadresse besteht immer aus den 4 Parametern. Als Trennzeichen wird ein Doppelpunkt verwendet. Dies wird im Listing 5.3 schematisch dargestellt.

Parameter	Beschreibung	Wertebereich
IPV4Adresse	Die IP Adresse des Modbus Server	lt. Netzwerk
SlaveID	Die Slave ID des Modbus Servers	1-247
Registeradresse	Die Registeradresse aus welcher ein Wert gelesen bzw. geschrieben werden soll	0-65535
SunSpecDatentyp	SunSpec Datentyp des Werts welcher gelesen bzw. geschrieben werden soll	

Tabelle 5.1: Komponenten der Remoteaddress

```
1 String Remoteaddress = "IPv4Adresse:SlaveId:  
Registeradresse:SunSpecDatentyp"
```

Listing 5.3: Zusammensetzung Remoteaddress

Der SunSpec Datentyp, welcher mit der *Remoteaddress* übergeben wird, ist folgendermaßen aufgebaut: *Datentyp/Länge*. Das Zeichen / ist hier im Text zur besseren Darstellung eingefügt worden. Die Länge ist bei primitiven Datentypen die Bitanzahl. Bei einem String ist dies die Zeichenanzahl. Da ein Holding Register immer 2 Zeichen enthält, wird die Länge welche dem String mitgegeben wird immer auf die nächsthöhere gerade Zahl aufgerundet, wenn ungerade. Die gerade Zahl dividiert durch zwei ergibt die benötigte Registeranzahl für den String. Folgende Aufzählung zeigt eine Übersicht der SunSpec Datentypen, die implementiert worden sind:

- int16
- uint16
- acc16
- enum16
- bitfield16
- pad16
- int32
- uint32
- acc32
- bitfield32
- int64
- acc64
- float32
- sunssf
- stringx: wobei $2 \leq x \leq 250$

5.6 Dataobject

In diesem Kapitel wird jenes Objekt beschrieben welches die Daten enthält, die von einem Server ausgelesen werden. Das Datenobjekt ist wie in Listing 5.4 beschrieben aufgebaut. Es besteht aus einem Objekt mit dem Namen “data“ und einem `SunSpecDataType` Objekt mit dem Namen `datatype`. In `data` sind die Daten welche zurückgegeben werden enthalten. Das Objekt `datatype` enthält den Datentyp welchen die Daten im Objekt `data` haben. Somit ist für die Klasse welche das `Dataobject` als Rückgabe erhält sichergestellt das die Daten in einen Java konformen Datentyp umgewandelt werden können.

```
1 public class DataObject {
2     Object data;
3     SunSpecDataType datatype;
4
5     public DataObject() {
6         this.data = new Object();
7     }
8 }
```

Listing 5.4: Dataobject

5.7 Result

Result ist ein Objekt, welches aus `Dataobject` 5.6 und `Status` 5.4 besteht. Dieses Objekt wird für die Methode `request` die im Abschnitt 5.10 beschreiben ist, benötigt. Listing 5.5 zeigt den Aufbau von `Result`.

```
1 public class Result {
2     DataObject data;
3     Status status;
4
5     public Result() {
6         this.data = new DataObject();
7     }
8 }
```

Listing 5.5: Result

5.8 ModbusTCPAdapter

Die Messaging Funktionen des Adapters sind in ModbusTCPAdapter als Methodenaufrufe realisiert. Die Klasse ist als Singleton implementiert. Die Entscheidung, den Adapter als Singleton auszuführen wurde unter dem Gesichtspunkt getroffen, dass es bei mehreren Adapter Objekten zur Überschneidung von Modbus Transaktionen kommen kann. Durch die Implementierung als Singleton wurde dies schon im Vorhinein ausgeschlossen. Somit kann das Connectivity Modul genau einen ModbusTCP Adapter besitzen. Mit der Methode ModbusTCPAdapter.getInstance() wird diese instanziiert bzw. gibt die Instanz zurück wenn bereits instanziiert.

5.8.1 Grundeinstellungen

In der ModbusTCPAdapter Klasse werden die Grundeinstellungen des Adapters getroffen. Diese sind in Tabelle 5.2 dargestellt. Die Variablen sind vom Attribut private. Daher werden Getter und Setter Methoden für diese Variablen bereitgestellt.

Variablenname	Datentyp	Standardwert	Beschreibung
TCPport	int	502	TCP Port des Modbus Servers
Timeout	int	100	Timeout einer Modbus Transaktion in Millisekunden. Innerhalb dieser Zeit muss die Antwort des Modbus Servers am Client eintreffen.
retries	int	1	Anzahl der Versuche einer Modbus Transaktion.
refresh	long	1000	Wird benötigt für das Polling bei Publish-Subscribe. Einheit in Millisekunden.

Tabelle 5.2: Grundeinstellungen Modbus TCP Adapter

5.8.2 ConnectionParameter

Wird ein Objekt der Klasse ConnectionParameter instanziiert, so dient dies als Datenhalteobjekt der Remoteaddress. Dabei wird der String vom Typ *Remoteaddress* in die einzelnen Komponenten zerlegt und in für die weiter verwendeten Funktionen gültige Datentypen gewandelt. Diese Aufgabe übernimmt die Methode SplitString in Abschnitt 5.8.3

5.8.3 SplitString

Die Methode *SplitString* ist private und wird daher nur von Methoden der Klasse *ModbusTCPAdapter* aufgerufen. Als Übergabeparameter wird ein String mit dem Format *Remoteaddress* benötigt. Die Methode zerlegt die *Remoteaddress* in die einzelnen Parameter und überprüft die Plausibilität der Werte. Ist ein Parameter ungültig, so wird eine Exception geworfen. Diese wird an die aufrufende Funktion weitergeleitet bis zum Aufruf der jeweiligen Methode der *ModbusTCPAdapter* Klasse, welche die Methode *SplitString* benötigt. Die Methode besitzt als Rückgabe ein Objekt vom Typ *ConnectionParameter*.

5.9 Push Funktion (send)

In der Implementierung wird die Push Funktion *send* genannt. Bei der Push 1:n Funktion schreibt ein Client den Wert eines Datenpunkts auf eine Anzahl von n Modbus Servern. Die Variable n muss größer gleich 1 sein. Als Übergabeparameter erhält die Methode eine Liste von *Remoteaddress* und ein Objekt vom Typ *DataObject*. Im Objekt *DataObject* ist es erforderlich das Objekt *data* mit den zu sendenden Werten zu beschreiben. Das Objekt *SunSpecDataType* benötigt keinen Wert, da der Datentyp des zu schreibenden Werts von der *Remoteaddress* bereitgestellt wird. Zurückgegeben wird eine Liste von *Status* Objekten. Das Listing 5.6 zeigt die Methode *send* mit den dementsprechenden Übergabe und Rückgabeparametern.

```
1 public List<Status> send(List<String> Remoteaddress,  
    DataObject Value) throws Exception{};
```

Listing 5.6: Push 1:n

Die *send* Methode wird folgendermaßen abgearbeitet: Im ersten Schritt wird die Liste an *Remoteaddress* Objekten überprüft. Diese muss mindestens einen Eintrag besitzen. Ist kein Eintrag vorhanden, dann wird im Rückgabeobjekt der Status *KeineVerbindungsparameter* geschrieben und die Methode wird abgebrochen. Ist mindestens ein Eintrag vorhanden, so wird dieser auf Plausibilität überprüft. Ist eine *Remoteaddress* nicht in Ordnung, so wird eine Exception geworfen. Diese wird an die *send* Methode weitergeleitet und muss dementsprechend beim Aufruf dieser Methode weitergeleitet oder gefangen werden. Ist die Liste an *Remoteaddress* Objekten gültig, dann werden die Einträge nacheinander abgearbeitet. Dazu wird zuerst ein Modbus Master Objekt erzeugt. Diesem werden die IP Adresse und der Port des Modbus Servers übergeben. Anschließend werden über Setter Methoden die Grundeinstellungen *Timeout* und *re-*

tries des *ModbusTCPAdapter* Objekts an das Modbus Master Objekt übergeben. Im nächsten Schritt wird eine TCP Verbindung zum Modbus Server aufgebaut. Dieser Methodenaufruf wird mit einem try/catch umgeben. Kann keine Verbindung aufgebaut werden, dann wirft die Methode eine Exception. Ist dies der Fall, wird in das Rückgabeobjekt der Methode *send*, der Status *ErrorVerbindungsaufbau* geschrieben und der nächste Eintrag in der Liste bearbeitet. Ist die Verbindung erfolgreich, dann wird die Methode zum Schreiben eines Wertes an einem Modbus Server aufgerufen. Dieser Methodenaufruf wird ebenfalls mit einem try/catch umgeben. Wirft die Methode eine Exception, dann wird in das Rückgabeobjekt der Methode *send* der Status *ErrorSenden* geschrieben. Ist der Methodenaufruf erfolgreich, dann wird der Status *WertGeschrieben* in das Rückgabeobjekt geschrieben. Nach Abarbeitung der Liste wird die Liste an *Status* Objekten von der Methode *send* zurückgegeben.

5.10 Request-Response Funktion (request)

In der Implementierung wird die Methode *request* genannt. Request-Response entspricht im Prinzip schon dem Funktionscode 0x03 (Lese Holding Registers) des Modbus Protokolls. Dabei wird ein Datenpunkt in den Registern abgefragt. Abhängig vom Datentyp des Datenpunkts auch mehrere Register. Listing 5.7 zeigt den Aufbau der Methode. Die *request* Methode benötigt als Übergabeparameter nur einen String vom Aufbau *Remoteaddress*. Als Rückgabewert wird ein Objekt vom Typ *Result* zurückgegeben. Diese Methode kann nur einen Datenpunkt an einem Modbus Server auslesen. Für mehrere Datenpunkte muss die Methode dementsprechend oft aufgerufen werden.

```
1 public Result request(String Remoteaddress) throws  
    Exception {}
```

Listing 5.7: Request-Response

Die Request Methode wird folgendermaßen abgearbeitet: Zuerst wird die Plausibilität des *Remoteaddress* Übergabeparameter geprüft. Bei erfolgreicher Prüfung wird eine TCP Verbindung zum Modbus Server aufgebaut. Bei erfolgreicher Verbindung wird der Datenpunkt, welcher durch die *Remoteaddress* spezifiziert, ist vom Modbus Server gelesen. Ist dies erfolgreich, dann wird der Wert in das Objekt welches zurückgegeben wird geschrieben. Der Status des Rückgabeobjekt wird auf *WertEmpfangen* gesetzt. Ist die Transaktion nicht erfolgreich, dann wird in das Rückgabeobjekt nur der Status *ErrorEmpfangen* geschrieben. Anschließend wird das Objekt vom Typ *Result* von der Methode zurückgegeben.

5.11 Publish-Subscribe Funktion (subscribe, unsubscribe)

Ein Modbus Server agiert nur passiv. Dieser kann keine Verbindung aktiv aufbauen und Daten an den Client senden. Somit ist es nicht möglich, Publish-Subscribe am Server zu implementieren. Um diese Funktion trotzdem zu realisieren, übernimmt diese Aufgabe der Protokolladapter durch polling. Dabei sendet der Protokolladapter Abfragen an den Modbus Server. Es wird überprüft ob in dem Register, welches am Modbus Server abonniert wurde eine Werteänderung stattgefunden hat. Bei einer Änderung wird der Subscriber benachrichtigt. In der Implementierung werden zwei Methoden bereitgestellt für Publish-Subscribe. Die Methode *subscribe*, welche einen neuen Subscriber erstellt. Diese Methode benötigt als Übergabeparameter einen String mit den Namen des Subscribers und einen weiteren String im Format *Remoteaddress*. Dies wird in Listing 5.8 dargestellt. Die Methode *unsubscribe* entfernt den Subscriber. Es werden wie bei der Methode *subscribe* dieselben Übergabeparameter benötigt. Siehe Listing 5.9. Die Idee, das Observer-Pattern zu benutzen um die Publish-Subscribe Funktion zu implementieren, wurde aus [?] und [?] übernommen.

```
1 public void subscribe(String subscriber, String
    Remoteaddress) throws Exception {}
```

Listing 5.8: subscribe

```
1 public void unsubscribe(String subscriber, String
    Remoteaddress) throws Exception {}
```

Listing 5.9: unsubscribe

5.11.1 Connectivity Modul

Das Connectivity Modul fordert am Protokolladapter einen neuen Subscriber an. Der Subscriber muss an das Connectivity Modul rückmelden, ob sich ein Wert geändert hat. Das Connectivity Modul ist ein Enum Singleton [?]. In der Implementierung wird dieses *ConnectivityModule* genannt. Dies ist in Listing 5.10 dargestellt. Die Klasse *ConnectivityModule* wird instanziiert, wenn ein Subscriber das erste mal die Methode *receive* aufruft. In der Methode *receive* muss dann programmiert werden, welches Modul im OpenNES System die Daten erhält. Die dafür notwendigen Informationen werden *receive* als Parameter übergeben.

```
1 public enum ConnectivityModule {  
    INSTANCE;  
3 public void receive(String subscriber, String  
    remoteAddress, DataObject data) {  
    //Write code here  
5 }  
}
```

Listing 5.10: Connectivity Modul als enum Singleton

5.11.2 Subscriber

Ein Objekt vom Typ *Subscriber* ist ein Datenhalteobjekt, welches die Informationen für genau einen Subscriber enthält. Weiters wird in der *Subscriber* Klasse das Interface *Observer* implementiert. Im *Observer* Interface sind die Getter und Setter Methoden definiert, welche in der Klasse *Subscriber* implementiert werden. In einem Objekt vom Typ *Subscriber* besitzen alle Variablen eine Getter Methode. Die Variablen *Value* und *newValue*, welche die aktuellen Werte enthalten die abonniert wurden, besitzen auch eine Setter Methode. Die übrigen Parameter des Objekts werden beim Konstruktoraufwurf von *Subscriber* übergeben. Während der Laufzeit können die Parameter eines Subscribers nicht geändert werden. Es muss ein neuer bzw. weitere Subscriber erzeugt werden, wenn eine Parameteränderung notwendig ist. Die Grundeinstellungen *TCPport*, *Timeout*, *retries*, werden aus dem *ModbusTCPAdapter* Singleton für jedes *Subscriber* Objekt übernommen, siehe Tabelle 5.2.

5.11.3 Observer

Im *Observer* Interface sind die Getter und Setter Methoden definiert, die auf die Variablen der *Subscriber* Klasse zugreifen. Das *Observer* Interface ist in der *Subscriber* Klasse implementiert. Somit wird die *Subscriber* Klasse entkoppelt, da nach der Erzeugung eines *Subscriber* Objekts mit dem *Observer* Objekt weitergearbeitet wird.

5.11.4 Grabber

In der *Grabber* Klasse ist der Hauptteil der Publish-Subscribe Funktion des Protokolladapters realisiert. Ein Objekt vom Typ *Grabber* wird bei der Instanziierung der Klasse *ModbusTCPAdapter* erzeugt. Das Interface *Runnable* ist in der Klasse *Grabber* implementiert. Dadurch kann das *Grabber* Objekt bei der Instanziierung von *ModbusTCPAdapter* in einem eigenen Thread gestartet werden. Zwei wichtige Methoden der *Grabber* Klasse sind *register* und *unregister*.

Die Methode *register* benötigt als Übergabeparameter ein Objekt vom Typ *Observer*. Da das *Observer* Interface in der Klasse *Subscriber* implementiert worden ist, ist es möglich, über diese auf ein *Subscriber* Objekt zu referenzieren. Die *Grabber* Klasse besitzt eine Liste von *Observer* Objekten. Wird die Methode *register* aufgerufen, so fügt diese ein *Observer* Objekt der Liste hinzu. Ein neuer Subscriber ist hinzugefügt.

Der Methode *unregister* wird als Übergabeparameter der String im Format *RemoteAddress* und ein String mit dem Namen des Subscribers übergeben. Die Methode durchsucht die Liste von *Observer* Objekten. Wird ein Element in der Liste gefunden, bei dem beide Parameter mit den Übergabeparametern übereinstimmen dann wird dieses aus der Liste entfernt. Auf das *Observer* Objekt verweist nun keine Referenz mehr. Folgender Abschnitt beschreibt die Funktionen in der Methode *run* der *Grabber* Klasse. Diese sind alle in der Endlosschleife *while(true)* enthalten. Im ersten Schritt wird überprüft, ob in der *Observer* bzw. *Subscriber* Liste mehrfach dieselben Einträge vorhanden sind. Ein Subscriber ist durch den Namen und die *Remoteaddress* definiert. Die *Remoteaddress* reicht aus, um einen abonnierten Datenpunkt eindeutig zu definieren. Da dem Subscriber auch ein Name vergeben wird, können mehrere Subscriber mit unterschiedlichen Namen denselben Datenpunkt abonnieren. Es wird daher aus der Liste von *Observer* Objekten eine Liste erzeugt, in der jede *Remoteaddress* nur einmal vorkommt.

Da die Publish-Subscribe Funktion durch polling realisiert ist, kann durch die Liste mit einzigartigen *Remoteaddress* Elementen, sofern es Subscriber gibt, welche dieselbe *Remoteaddress* besitzen, die Anzahl der Abfragen an einen Server reduziert werden. Im nächsten Schritt wird der Wert von allen Modbus Servern abgefragt. Die Liste von *Remoteaddress* Elementen wird sequenziell abgearbeitet. Das Abfragen der Werte funktioniert gleich wie bei Request-Response. Deshalb sei hier auf das Kapitel 5.10 verwiesen. Der empfangene Wert wird in die Variable *newValue* bei allen *Observer* Objekten, die die *Remoteaddress* besitzen geschrieben. Nachdem die *Remoteaddress* Liste abgearbeitet ist, wird durch die Liste von *Observer* Objekten iteriert. Dabei werden die Variablen *value* und *newValue* verglichen. Sind die Werte unterschiedlich, wird die Methode *receive* des Singleton *ConnectivityModule* aufgerufen. Anschließend wird der Wert der Variable *newValue* in die Variable *value* geschrieben.

Nach Abarbeitung der *Observer* Liste wird der Thread für eine Zeit pausiert. Diese Zeit ist in der Variable *refresh* des *ModbusTCPAdapter* Singleton vorgegeben. Über die Setter Methode *setRefresh* kann diese geändert werden, siehe Tabelle 5.2.

6 Funktionstest/Validierung

6.1 Virtual Machine Management

Zum Testen des Protokolladapters ist ein Netzwerk von **VM!s** (**VM!s**) eingerichtet worden. Als Virtualisierungssoftware wurde Virtualbox 5.1¹ von Oracle verwendet. Genauere Informationen zur VM sind in der Tabelle 6.1 enthalten. Als nächster Schritt folgte die Installation der Entwicklungsumgebung Eclipse. Abbildung 6.1 zeigt eine Übersicht des eingerichteten Netzwerks. Dabei agiert eine VM als Modbus Client und zwei weitere VMs sind Modbus Server. Um die in Kapitel 5.9 beschriebene Funktion “Push 1:n“ zu simulieren sind mehrere Server notwendig. Die VMs sind über das interne Netzwerk verbunden, welches von Virtualbox bereitgestellt wird.

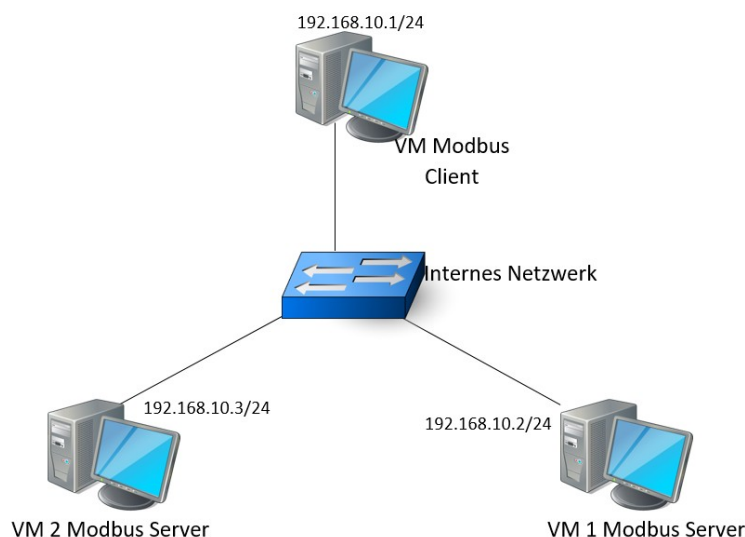


Abbildung 6.1: Netzwerkübersicht

	Client	Server 1	Server 2
IP V4 Adresse	192.168.10.1/24	192.168.10.2/24	192.168.10.3/24
OS	Windows 7 Ultimate 64 bit V6.1.7601 SP1 Build 7601		
RAM	2048Mb		
CPUs	4 paravirtualisiert		
Virtualisierungssoftware	Virtual Box 5.1.12 r112440		

Tabelle 6.1: Übersicht VMs

¹<https://www.virtualbox.org/>

6.2 Ausgangsbedingungen und Abgrenzungen

Als Testumgebung wurde die bereits in Kapitel 6.1 beschriebene Konfiguration verwendet. Auf der Client VM sind die Pakete mit dem Programm Wireshark² aufgezeichnet worden. Zur besseren Übersicht sind die Daten des TCP Segments welche für Modbus relevant sind in Tabellenform dargestellt. Die Publish-Subscribe Funktion wird nicht separat behandelt, da diese der Grundfunktion der Request-Response Funktion entspricht.

6.2.1 Modbus Server

Zum Testen der Funktionen des Protokolladapters werden zwei Modbus Server verwendet. Programmiert wurden diese unter der Verwendung der Java Library Modbus4j. SunSpec lässt verschiedene Informationsmodelle zu. Zum Testen wurde das in [?] auf den Seiten 25 bis 29 angegebene Modell implementiert. Im ersten Schritt wurden die im Dokument beschriebenen Register erzeugt und mit zufälligen Werten initialisiert. Das Objekt in welchem sich die Register befinden ist vom Typ *BasicProcessImage*. Beim Erzeugen des Objekts muss die Modbus Slave ID übergeben werden. Somit können auf einem Server mehrere Objekte vom Typ *BasicProcessImage* erzeugt werden. Die unterschiedlichen Objekte werden von einem Client über die Slave ID des Modbus Protokolls angesprochen. Bei Modbus TCP ist ein Server über die IP-Adresse und Slave ID eindeutig identifizierbar. Ein Modbus TCP Gerät kann deshalb mehrere Modbus Server bereitstellen. Es ist das Gateway zu mehreren Modbus Servern. Ein *BasicProcessImage* Objekt dient zum Austausch der Daten zwischen Modbus Server und der Applikation welche am Server läuft. Nach der Erzeugung des Prozessabbildes wird der Modbus Server in einem Thread gestartet. Parallel dazu wird ein weiterer Thread gestartet. Dem Programm im zweiten Thread wird die Referenz des *BasicProcessImage* Objekts übergeben. Das Programm in diesem Thread springt in eine Endlosschleife und gibt die Werte der Register in der Kommandozeile aus. Bei statischen Werten ist es nicht möglich, die Funktion Publish-Subscribe zu testen. Deshalb ändert dieses Programm die Werte in den Registern. Nach dem Durchlauf des Programms wird der Thread für eine Zeit pausiert.

²<https://www.wireshark.org/>

6.3 Push1-N (send)

In diesem Kapitel wird der Test zum Validieren der implementierten Push1-N Funktion beschrieben.

6.3.1 Ausgangssituation

Auf VM1 ist ein Modbus Server eingerichtet. Im Bild 6.2 ist dargestellt, wie die Applikation zyklisch den SunSpec Float Wert ab Registeradresse 40072 ausgibt.

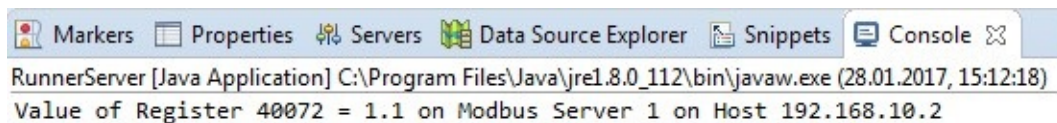


Abbildung 6.2: Modbus Server VM1 vor dem Test

Auf VM2 sind zwei Modbus Server eingerichtet. Im Bild 6.3 ist dargestellt, wie die Applikation zyklisch den SunSpec Float Wert ab Registeradresse 40072 der beiden Modbus Server ausgibt.

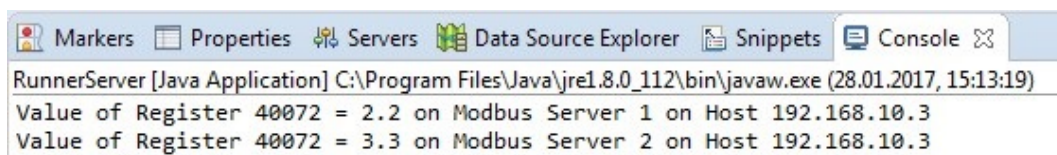


Abbildung 6.3: Modbus Server VM2 vor dem Test

Angemerkt sei, dass alle 3 Register vor dem Test einen unterschiedlichen Wert besitzen.

6.3.2 Test

Mit der *send* Methode der *ModbusTCPAdapter* Klasse wird nun ein Wert an die im Kapitel 6.3.1 beschriebenen Server gesendet. Listing 6.1 zeigt den Methodenaufruf. Dabei wird eine Liste mit der *Remoteaddress* der 3 Modbus Server erzeugt. Ebenfalls wird eine Liste an *Status* Objekten erzeugt. Geschrieben soll der Wert 10.0 im SunSpec *Float* Format werden. Nach dem Aufruf der Methode werden die Rückgabeobjekte der Methode zu Testzwecken auf die Kommandozeile geschrieben. Im folgenden Abschnitt wird, stellvertretend für alle Transaktionen, nur eine Transaktion zu einem Server betrachtet. Prinzipiell funktionieren die Transaktionen gleich.

```
public static void main(String[] args){
2  //Neuer Adapter
    ModbusTCPAdapter Adapter = ModbusTCPAdapter.
        getInstanz();

4
    // Liste von Verbindungen
6    List<String> ConnectionParameter = new LinkedList
        <>();
    //Liste fuer Rueckmeldungen
8    List<Status> FeedbackStatus = new LinkedList<>();

10   //Wert zuweisen
    DataObject value = new DataObject();
12   value.data = 10.0;

14   //Modbus Server hinzufuegen
    ConnectionParameter.add("192.168.10.2:1:40072:
        float32");
16    ConnectionParameter.add("192.168.10.3:1:40072:
        float32");
    ConnectionParameter.add("192.168.10.3:2:40072:
        float32");

18
    //push Funktion
20    try {
        FeedbackStatus = Adapter.send(ConnectionParameter,
            value);
22    } catch (Exception e) {
        // TODO Auto-generated catch block
24        e.printStackTrace();
    }

26
    //Ausgabe der Rueckmeldungen in die Commandline
28    for(int i = 0; i < FeedbackStatus.size(); i++){
        System.out.println("Feedback Server " + i + " " +
            FeedbackStatus.get(i));
30    }
}
```

Listing 6.1: Methodenaufruf send

Die Tabelle 6.2 zeigt die Daten des TCP Segments der Modbus Anfrage an den Modbus Server der VM1. Der Aufbau eines Modbus Frames wurde in Kapitel 3.1.3 beschrieben. Im folgenden Text wird dies anhand der aufgezeichneten Daten validiert. Die Felder des MBAP Headers sind bereits in Kapitel 3.1.3 beschrieben. Als Einziges soll hier das letzte Byte im MBAP Header betrachtet werden, welches den Wert 1 hat. Dies ist die Slave ID, welche übereinstimmt mit jener die der *send* Methode übergeben wurde. Als Funktionscode hat die Modbus Library 0x10 gewählt. Dieser wurde in Kapitel 3.1.5 beschreiben. Die Startadresse mit dem Wert 40072 stimmt mit dem Übergabeparameter der *send* Methode überein. Es werden 2 Register für einen SunSpec Float benötigt. Dies stimmt mit dem Wert in der Tabelle überein. Die Anzahl an Werten beträgt 4. Die 4 Registerwerte umgerechnet ergeben die Gleitkommazahl 10.0, welche dem Übergabeparameter der Methode *send* entspricht.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0	0	0 = Modbus Protokoll	
3	0			
4	0	11	Anzahl der darauffolgenden Byte	
5	0B			
6	1	1	Slave ID	Daten
7	10	16	Funktionscode: Schreibe Multiple Registers	
8	9C	40072	Start Registeradresse der zu schreibenden Daten	
9	88			
10	0	2	Anzahl der Register welche geschrieben werden	
11	2			
12	4	4	Anzahl der Werte	
13	41	10.0	Registerwerte: Diese ergeben umgerechnet die Gleitkommazahl 10.0	
14	20			
15	0			
16	0			

Tabelle 6.2: Modbus Request der Push Funktion

Die Tabelle 6.3 stellt die Antwort des Modbus Servers der VM1 dar. Vergleicht man diese mit der zu erwartenden (siehe 3.1.5) und der an den Server gesendeten Anfrage (siehe 6.2) kommt man zu folgenden Ergebnis. Es wird der an den Server gesendete Funktionscode wieder zurückgesendet. Dies bedeutet die Modbus Transaktion war erfolgreich. Die Register Startadresse und die Anzahl der Register wurden wie im Modbus Standard definiert zurückgesendet und stimmen auch mit den gesendeten Daten überein.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0	0	0 = Modbus Protokoll	
3	0			
4	0	6	Anzahl der darauffolgenden Byte	
5	6			
6	1	1	Slave ID	
7	10	16	Funktionscode: Schreibe Multiple Registers	Daten
8	9C	40072	Start Registeradresse der zu schreibenden Daten	
9	88			
10	0	2	Anzahl der Register welche geschrieben werden	
11	2			

Tabelle 6.3: Modbus Response der Push Funktion

Im zweiten Testversuch der *send* Methode wurde versucht den Wert in einem Register zu ändern, welches nicht vorhanden ist. Tabelle 6.4 zeigt die Antwort des Modbus Servers. Der Funktionscode oder nun Errorcode besitzt den Wert 0x90. Dies entspricht dem zu erwartenden Code laut Modbus Standard. Als weiterer Wert wird noch ein Exceptioncode mitübertragen. Dieser besitzt den Wert 0x02 und entspricht ebenfalls Modbus Spezifikation.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0			
3	0	0	0 = Modbus Protokoll	
4	0			
5	3	3	Anzahl der darauffolgenden Byte	
6	1	1	Slave ID	Data
7	90		Fehler Code	
8	2		Exceptioncode	

Tabelle 6.4: Fehlerhafte Modbus Response der Push Funktion

6.3.3 Testende

Abbildungen 6.4 und 6.5 zeigen die Ausgabe des Float Wertes, welcher während des Test geschrieben wurde. Wie aus den Bildern ersichtlich wurde der Wert 10.0 in die Register der 3 Server geschrieben.

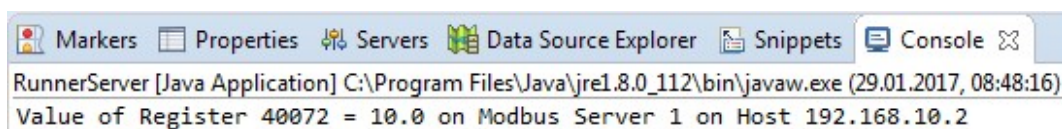


Abbildung 6.4: Modbus Server VM1 nach dem Test

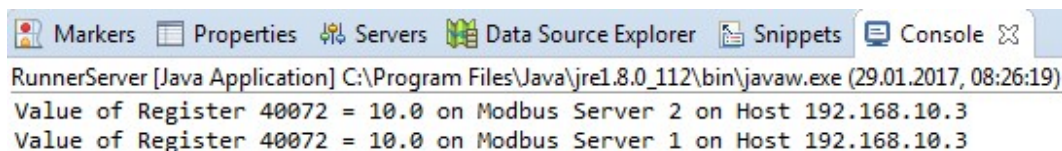


Abbildung 6.5: Modbus Server VM2 nach dem Test

6.4 Request-Response (request)

In diesem Kapitel wird der Test zum Validieren der implementierten Request-Response Funktion beschrieben.

Ausgangssituation

Die Ausgangssituation der Request-Response Funktion ist dieselbe wie beim Test der Push Funktion, siehe Kapitel 6.3.1. Für diesen Test wird nur der Modbus Server der VM1 benötigt.

6.4.1 Test

Mit der *request* Methode der *ModbusTCPAdapter* Klasse wird nun ein Datenpunkt des Modbus Servers der VM1 ausgelesen. Listing 6.2 zeigt den Methodenaufruf. Der Methode wird der String im Format *Remoteaddress* übergeben. Rückgegeben wird ein Objekt *Result*. Bei diesen Test soll der SunSpec *Float* Wert beginnend ab Register 40072 ausgelesen werden. Nach dem Aufruf der Methode wird das Rückgabeobjekt der Methode zu Testzwecken auf die Kommandozeile geschrieben. Hier soll auch der abgefragte Wert ausgegeben werden.

```
1 public static void main(String[] args) {  
    //Neuer Adapter  
3   ModbusTCPAdapter Adapter = ModbusTCPAdapter.  
        getInstance();  
    //request response  
5   Result Feedback = new Result();  
    try {  
7       Feedback = Adapter.request("192.168.10.2:1:40072:  
            float32");  
    } catch (Exception e) {  
9       // TODO Auto-generated catch block  
        e.printStackTrace();  
11    }  
    //Ausgabe der Antwort der Request-Response  
13    System.out.println(Feedback.data.data.toString());  
    System.out.println(Feedback.status);  
15    System.out.println("Datentyp " + Feedback.data.  
        datatype);  
}
```

Listing 6.2: Methodenaufruf request

Tabelle 6.5 zeigt die Daten des TCP Segments der Modbus Anfrage an den Modbus Server der VM1. Der Aufbau des Modbus Frames wird im Kapitel 3.1.3 beschrieben. Im folgenden Text wird dies anhand der aufgezeichneten Daten validiert. Die Felder des MBAP Headers sind bereits in Kapitel 3.1.3 beschrieben. Als einziges soll hier das letzte Byte, die Slave ID im MBAP Header, betrachtet werden welches den Wert 1 hat. Diese stimmt mit jener überein, die der *request* Methode übergeben wurde. Der im Kapitel 3.1.5 beschriebene Funktionscode 0x03 wurde durch die Modbus Library vergeben. Die Startadresse mit dem Wert 40072 stimmt mit dem Übergabeparameter der *request* Methode überein. Es werden 2 Register für einen SunSpec *Float* Wert benötigt. Auch dies wird in der Tabelle so dargestellt.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0	0	0 = Modbus Protokoll	
3	0			
4	0	6	Anzahl der darauffolgenden Byte	
5	6			
6	1	1	Slave ID	Data
7	3	40072	Funktionscode	
8	9C		Start Register Adresse der zu lesenden Daten	
9	88			
10	0	2	Anzahl der Register welche gelesen werden	
11	2			

Tabelle 6.5: Modbus Request der Request-Response Funktion

Tabelle 6.6 stellt die Modbus Antwort des Modbus Servers der VM1 dar. Vergleicht man diese mit der zu erwartenden (siehe 3.1.5) und der an den Server gesendeten Anfrage siehe Tabelle 6.5 kommt man zu folgenden Ergebnis. Es wird der an den Server gesendete Funktionscode wieder zurückgesendet. Dies bedeutet die Modbus Transaktion war erfolgreich. Es wird die Anzahl an Werten zurückgesendet und die Werte in den Registern. Diese 4 Byte ergeben umgerechnet die Gleitkommazahl 1.1. Dieser Wert wird entsprechend Abbildung 6.2 im Register dargestellt.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0	0	0 = Modbus Protokoll	
3	0			
4	0	7	Anzahl der darauffolgenden Byte	
5	7			
6	1	1	Slave ID	
7	3	3	Funktionscode	
8	4	4	Anzahl der Werte	
9	3F	1.1	Gelesene Register Werte: Diese ergeben umgerechnet die Gleitkommazahl 1.1	Daten
10	8C			
11	CC			
12	CD			

Tabelle 6.6: Modbus Response der Request-Response Funktion

Im zweiten Testversuch der *request* Methode wurde versucht, den Wert in einem Register des Modbus Servers der VM1 welches nicht vorhanden ist auszulesen. Tabelle 6.7 zeigt die Antwort des Modbus Servers. Der Funktionscode oder nun Errorcode besitzt den Wert 0x83. Dies entspricht dem zu erwartenden Code laut Modbus Standard. Als weiterer Wert wird noch ein Exceptioncode mitübertragen. Dieser besitzt den Wert 0x02 und entspricht ebenfalls der Modbus Spezifikation.

TCP Daten				
Byte	hex	Wert	Beschreibung	
0	0	0	Identifizierung der Modbus Request / Response Transaktion	MBAP Header
1	0			
2	0		0 = Modbus Protokoll	
3	0	3		
4	0		Anzahl der darauffolgenden Byte	
5	3			
6	1	1	Slave ID	Daten
7	83		Funktionscode	
8	2		Exceptioncode	

Tabelle 6.7: Fehlerhafte Modbus Response der Request-Response Funktion

7 Fazit

Der in dieser Arbeit entworfene **PA!** verbessert die Kommunikation zwischen OpenNES und Legacy Geräten. Mit Hilfe des in Punkt 6 beschriebenen Testszenarios wurde überprüft, ob der Adapter die in den Anforderungen spezifizierten Funktionen erfüllt. Die Anforderungen werden erfüllt und er kann SunSpec Datenpunkte lesen und schreiben. Sollte es notwendig sein weitere Modelle zu implementieren, z.B.: andere als die von SunSpec definierten Datentypen, ist dies problemlos möglich. Die Grundfunktionen des Adapters sind vorhanden und müssen gegebenenfalls nur angepasst werden.

Ein möglicher nächster Schritt, wäre es einen Modbus RTU Protokolladapter zu entwickeln. Wie in der Arbeit beschrieben, unterscheiden sich Modbus TCP und RTU prinzipiell nur durch das Übertragungsprotokoll. Die bereits für Modbus TCP verwendete Library Modbus4j unterstützt laut Dokumentation auch Modbus RTU. Weiters ist uns bei der Implementierung des Adapters und beim Durcharbeiten des Modbus Standards aufgefallen, dass das Modbus Protokoll keine Security Mechanismen eingebaut hat. Ein Modbus Client muss sich nicht beim Server authentifizieren bzw. benötigt keine Freigabe vom Server, um Daten zu lesen oder zu schreiben. Sobald sich also ein Client im selben Netzwerk wie der Server befindet, hat der Client vollen Zugriff auf den Server. Hierfür sei auf [?] verwiesen, wo die Erstellung eines Sicheren Modbus Protokolls behandelt wird. Zusammenfassend kann gesagt werden, dass der **PA!** eine relativ einfache **IKT!**-Lösung zur Einbindung von Standard Geräten in das OpenNES System ist.