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

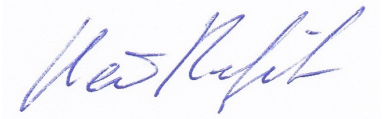
Studiengangsleiter:	FH-Prof. DI Dr. Gerhard Jöchtl
Betreuer:	DI Norbert Egger BSc

Salzburg, September 2018

Eidesstattliche Erklärung

Ich versicheren 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 interoperabe 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 beeing developed jointly by the Austrian Intitute 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

PWA Progressive Web Aplication

SP Smart Phone

SP Smart Phone

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Durch die Markteinführung des Smart Phone hat sich in unserem Leben geändert. Nicht nur unsere Kommunikation wurde dadurch verändert sondern unser Leben im allgemeinen, ist durch dieses kleine Wundergerät verändert worden. Wir haben fast ständig das SP im Einsatz um zu organisieren, zu spielen,zum Musik hören, und um unsere Kontakte zu 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.

1.1 Motivation

Wo werden die Apps am meisten verwendet

1.2 Zielsetzung

2 Grundlagen

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 ersetzten?".

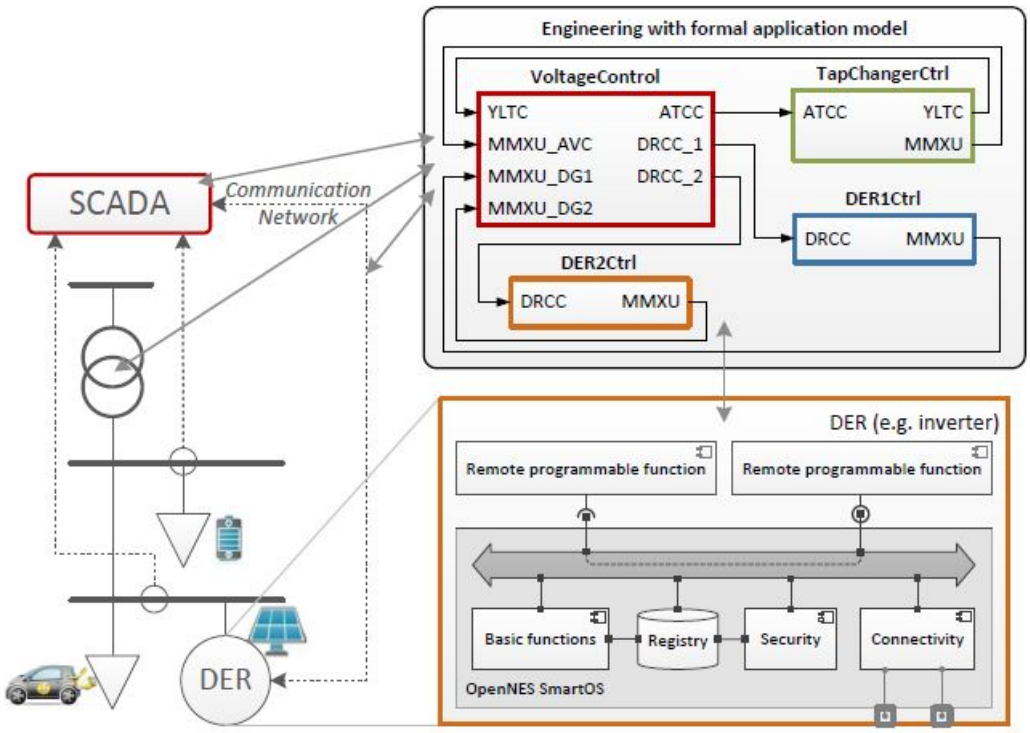


Abbildung 2.1: Übersicht des OpenNES Konzepts [**?**]

2.1 Geschichte Applikationsentwicklung

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

2.2 Mobile APP

2.2.1 Native Apps

2.2.2 Webapplikationen

2.2.3 Progressive Web Apps

2.2.4 Unterschiede zwischen Web Apps PWA und Native Apps

3 Basistechnologien

3.1 Manifest

In diesem Kapitel werden die Besonderheiten einer PWA erklärt.

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

3.1.2 Grundlagen

- Modbus TCP/IP

3.1.3 Protokollaufbau

3.1.4 Modbus Datenmodell

3.1.5 Funktionscodes

Lese Holding Registers

Schreibe Multiple Registers

3.1.6 ModBus TCP/IP

3.2 SunSpec

3.2.1 Übersicht

3.2.2 SunSpec Informationsmodell

3.2.3 SunSpec Datentypen

3.2.4 Register Mapping

4 Entwurf

In diesem Kapitel wird das Adaptermuster im Allgemeinen und die Anforderungen bzw. die Umsetzung des **PA!**s betrachtet. Wie in Kapitel **??** 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).*

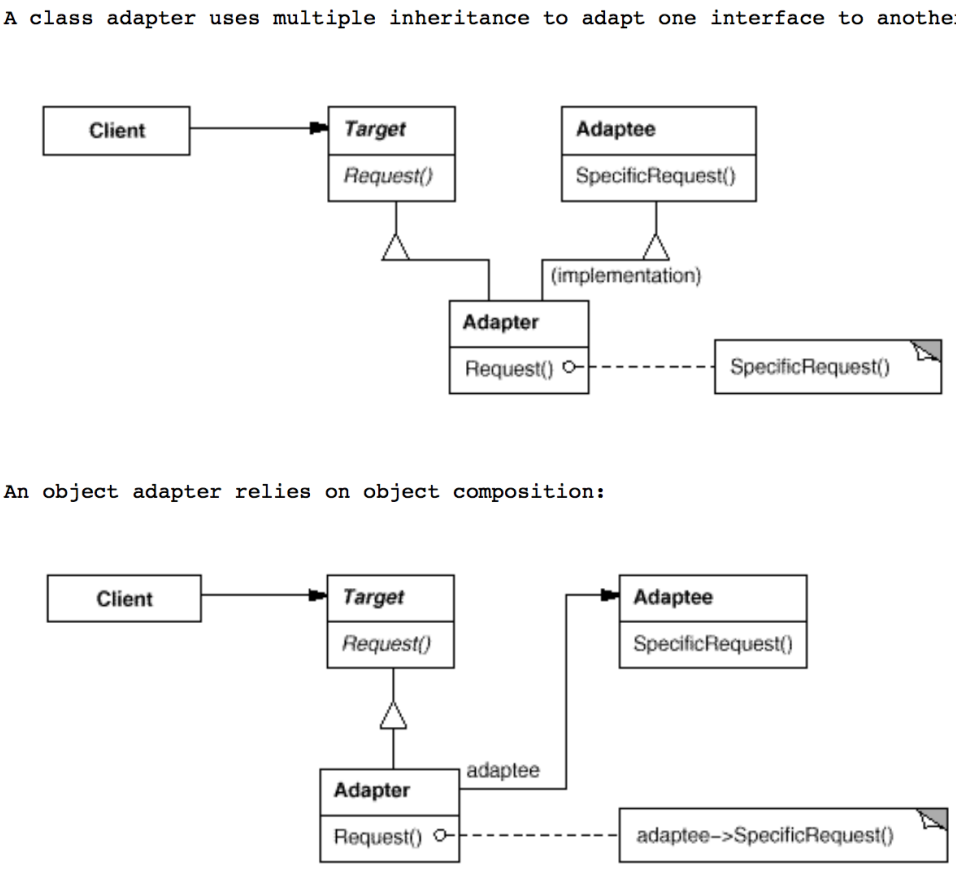


Abbildung 4.1: UML Diagramm des Objekt- bzw. Klassenadapters [**?** , S.159]

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 **??** 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 (**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 =
OpenNESAdddress ":" "{" 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 Modbus4j[1] 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 Remoteaddresse 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 <= x <= 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 konformem 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.getInstanz() 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 Connectivty 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 Konstruktoraufruf 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 5.1[1] 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.

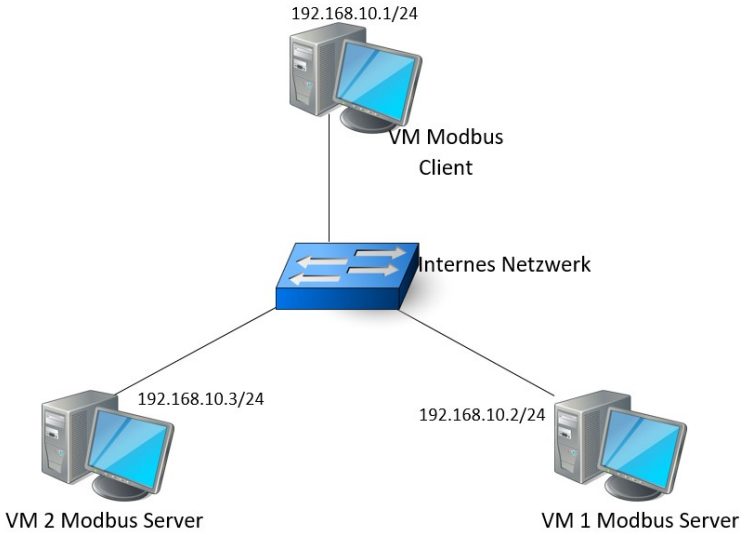


Abbildung 6.1: Netzwerkübersicht

	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 Wireshark[2] 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.

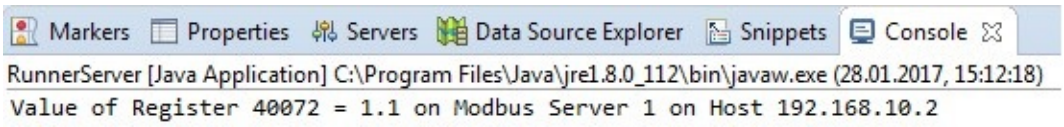


Abbildung 6.2: Modbus Server VM1 vor dem Test

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.

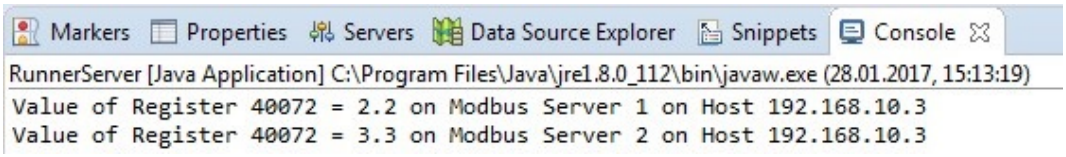


Abbildung 6.3: Modbus Server VM2 vor dem Test

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){
  //Neuer Adapter
  ModbusTCPAdapter Adapter = ModbusTCPAdapter.
     getInstanz();

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

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

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

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

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

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	
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			
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.

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

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	0	0 = Modbus Protokoll	
3	0			
4	0	3	Anzahl der darauffolgenden Byte	
5	3			
6	1	1	Slave ID	
7	90		Fehler Code	Data
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.

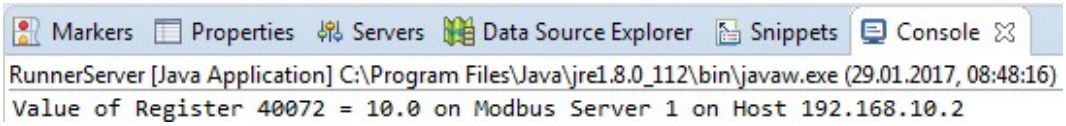


Abbildung 6.4: Modbus Server VM1 nach dem Test

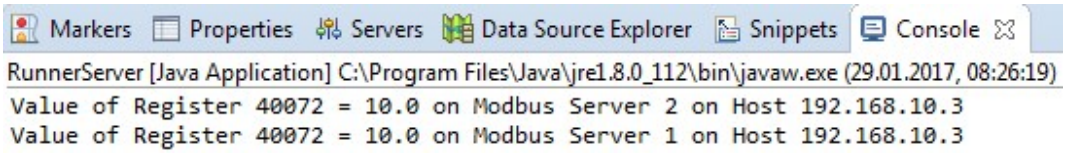


Abbildung 6.5: Modbus Server VM2 nach dem Test

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 wir 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.
       getInstanz();
    //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.

<table>
<tr><td colspan="5">TCP Daten</td></tr>
<tr><td>Byte</td><td>hex</td><td>Wert</td><td>Beschreibung</td><td></td></tr>
<tr><td>0</td><td>0</td><td rowspan="2">0</td><td rowspan="2">Identifizierung der Modbus
Request / Response Transaktion</td><td rowspan="6">MBAP Header</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>2</td><td>0</td><td rowspan="2">0</td><td rowspan="2">0 = Modbus Protokoll</td></tr>
<tr><td>3</td><td>0</td></tr>
<tr><td>4</td><td>0</td><td rowspan="2">6</td><td rowspan="2">Anzahl der daraufolgenden Byte</td></tr>
<tr><td>5</td><td>6</td></tr>
<tr><td>6</td><td>1</td><td>1</td><td>Salve ID</td></tr>
<tr><td>7</td><td>3</td><td></td><td>Funktionscode</td><td rowspan="5">Data</td></tr>
<tr><td>8</td><td>9C</td><td rowspan="2">40072</td><td rowspan="2">Start Register Adresse der
zu lesenden Daten</td></tr>
<tr><td>9</td><td>88</td></tr>
<tr><td>10</td><td>0</td><td rowspan="2">2</td><td rowspan="2">Anzahl der Register
welche gelesen werden</td></tr>
<tr><td>11</td><td>2</td></tr>
</table>

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	Daten
8	4	4	Anzahl der Werte	
9	3F	1.1	Gelesene Register Werte: Diese ergeben umgerechnet die Gleitkommazahl 1.1	
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	0 = Modbus Protokoll	
3	0			
4	0	3	Anzahl der darauffolgenden Byte	
5	3			
6	1	1	Slave ID	
7	83		Funktionscode	Daten
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.