THESIS

A DOMAIN-PROTOCOL MAPPING BASED MIDDLEWARE FOR DISTRIBUTED APPLICATION DEVELOPMENT

Submitted by

Sai Pradeep Mandalaparty

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2014

ABSTRACT

A DOMAIN-PROTOCOL MAPPING BASED MIDDLEWARE FOR DISTRIBUTED APPLICATION DEVELOPMENT

Distributed systems such as Internet of Things, Sensor Networks and Networked Control Systems are being used in various application domains, including industrial, environmental, medical and energy management domains. A distributed application in these domains may need to access data from different devices, where they may all be of the same type or a combination of different types. In addition, these devices may communicate through standardized protocols or proprietary interfaces. The development of such a distributed application may also require a team of developers with expertise in different disciplines. Therefore, the application development that involves heterogeneous devices and multidisciplinary teams can be made more effective by introducing an interface layer that shields developers from aspects of software and hardware heterogeneity.

This work proposes a 'domain-protocol mapping' technique that is implemented as a middleware framework. The proposed mapping method maps the application data schema represented as object-oriented domain object to the appropriate communication protocol packet data and also updates the domain object from the response packet data. The middleware provides APIs for the domain experts to read the data from the device or to write the data to the device. The marshalling and unmarshalling process of the domain objects are hidden from the domain expert who may or may not be a software engineer. The use of the developed middleware is illustrated in two case-studies, one involving a simulation of distributed network controls for power system and the other involving integration of different types of power meters in power monitoring application.

ACKNOWLEDGMENTS

First and foremost, I am extremely grateful to my supervising advisor, Professor Robert France, who provided knowledge, guidance, and support through some of the tough and challenging times. I am also grateful to the members of my dissertation committee Professor Sanjay Rajopadhye and Professor Peter Young for their comments and suggestions in critiquing the final draft of this dissertation.

In particular, I would specifically like to thank Mr. Jerry Duggan for helping me out with different aspects of the distributed application development problems and guiding me all the way through the completion of my research. His patience and expertise have helped me a great deal in designing the Middleware framework. I would like to thank all the faculty and staff of the Computer Science Department for making my graduate study pleasant.

I wish to thank my friends in Fort Collins, who I regard as family, for making my stay a memorable one. Finally, I would like to express my gratitude to my family members for their unconditional love and support and for giving me courage and motivation through the difficult and enjoyable times.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Problem Overview

Advances in distributed computing have led to the emergence of several distributed systems such as Internet of Things (IoT) [3], Sensor Networks (SN) and Networked Control Systems (NCS) [4]. These distributed systems are being used in various industrial applications, environmental applications, medical and energy management applications. Such distributed systems comprises *physical systems*, *sensors*, *actuators* and *control systems*. Sensors and actuators operate on the physical systems and the control systems maintain or control the physical systems indirectly through sensors and actuators. These entities are equipped with varying computing power, sensing and communication capabilities. They measure different physical/simulated conditions such as temperature, pressure, motion, humidity and power. These measured conditions are processed by domain applications which react, for example, by implementing post-processing algorithms, triggering alarms, activating the actuators or by applying business rules. Hereafter, in this thesis, IoT, SN, NCS will be referred to as distributed systems and devices refer to things, sensors, actuators and control systems.

Use of entities such as 'things' in IoT, 'sensors' in SN and 'sensors, actuators and control systems' in NCS come with new computer science challenges [5]. In particular, a given application may need to access different types of devices or devices of the same type, but managed by proprietary software and communicate through standard/proprietary protocols. The former is referred as *device heterogeneity* and the latter as *software heterogeneity*. To develop the applications which address device and software heterogeneity in distributed systems, there is a need to understand the following device characteristics:

- Device capabilities can range from very simple, limited resource sensor nodes to very capable and expensive systems.

- Based on the level of hardware/software accessibility of the devices, they can be classified as *Open* or *Closed*. Open devices allow the user to develop/modify the device functionality (i.e., the firmware) and the data exchange communication protocol, whereas closed devices do not allow modification of their functionality but provide access to the functionality through a fixed set of standard data exchange communication protocols and proprietary interfaces.

- Commercial Off-The-Shelf (COTS) devices, by definition, are closed devices. Vendors seek to differentiate their device functionality, yet seek to minimize investment by providing the access to the device functionality through standard communication protocols like Modbus [6], DNP3 [7] and SNMP [8] to query application data schema. In this context, application data schema corresponds to the data provided by the device.

Along with these device characteristics, the development of few distributed applications can also need a team of developers with different expertise. For instance, in the case of a multidisciplinary team investigating 'Distributed Networked Controls for Power System (NCS) application [1]'. The team consists of Electrical Engineers, Power Systems Engineers, and Software Engineers. The nature of the software written by the team consists of data capture from multiple devices, the control algorithm implementation, and validation of the control algorithms on simulated power systems. The domain experts, in this context, the power system engineers or electrical engineers, may not be specialized to implement different aspects of the above-mentioned device characteristics in the NCS application. Therefore, there is a need to separate the concerns of a multidisciplinary team by providing the appropriate APIs (Application Program Interfaces) between different roles in a distributed application development process.

Application development which involves support for heterogeneous devices and multidisciplinary teams can be simplified by careful separation of application data schema and device communication logic via protocol adapters by introducing an interface layer, referred to as middleware in the literature. A protocol adapter provides APIs to transform an application data schema to a protocol packet or update an application data schema from the protocol packet. In the context of this work, the challenges such a middleware should address are as follows:

- Leverage the investment in protocol adapter development among many applications.
- Separate the concerns of application developers (domain experts)from that of the protocol adapter developers (protocol engineers).
- Provide a mechanism to represent the application data schema in an abstract manner and to query the data from the devices via standard/proprietary protocols.

This work proposes a 'domain-protocol mapping' method that is implemented as middleware framework, to address the above-described challenges in distributed application development involving multidisciplinary teams and heterogeneous devices. The developed middleware is demonstrated by two case-studies, in the first case-study the middleware is used as a software layer in multiple client communications in 'Distributed Networked Controls for a Power System Case Study [1].' The second case study describes a scenario in building energy management, where the application data schema is fixed, and the middleware provides access to the ION [9] and ETON [10] power meter devices via a standard protocol. Security, privacy, device discovery aspects of the middleware are out of the scope of this thesis.

1.2 Overview of the solution

The proposed 'domain-protocol mapping' technique based middleware is conceptually similar to that of the hibernate [11] object-relational mapping (ORM) solution for Java. Hibernate provides APIs to map Java classes to database tables that also involve type conversion between Java and SQL data types, which relieve the software developer from common data persistence related programming tasks. Similarly, the proposed middleware maps application data schema, represented as object-oriented domain objects, to appropriate communication protocol packet data. Moreover, the middleware updates the domain objects from the response packet data relieving the domain experts from addressing numerous device and the communication protocol implementation details. An overview of the mapping technique based middleware architecture is illustrated in Fig 1.1. The layers of the architecture are described below:

Domain Object Layer: The Domain Object layer is responsible for defining the application data schema, which is represented as object-oriented domain objects. These objects encapsulate the data that a domain user wants to read from a device or write to a device.

Protocol Adapter Layer: The Protocol Adapter layer is responsible for encoding and formatting the domain objects into specific protocol packets based on protocol specific formatting rules. This layer also has additional components to process the received protocol packets and update the domain objects.

Device Metadata Layer: The Device Metadata layer describes the meta-information about the data provided by a device. The meta-information includes the type and the encoding format of the data, address location (memory map) of the data or the byte offset of the data in the protocol packet, and the description of the data provided by the device.

Mapping Layer: The Mapping layer is responsible for providing mapping specification between the domain object and the device meta-data. This mapping specification is used by a protocol adapter to transform the domain objects into the protocol packets and to update the domain objects from the protocol packets. The mapping is specific for a device-protocol combination.

Application Layer: The Application layer consists of different domain applications which queries the concerned application data schema from the domain object layer and act on the data accordingly by computing post-processing algorithms.

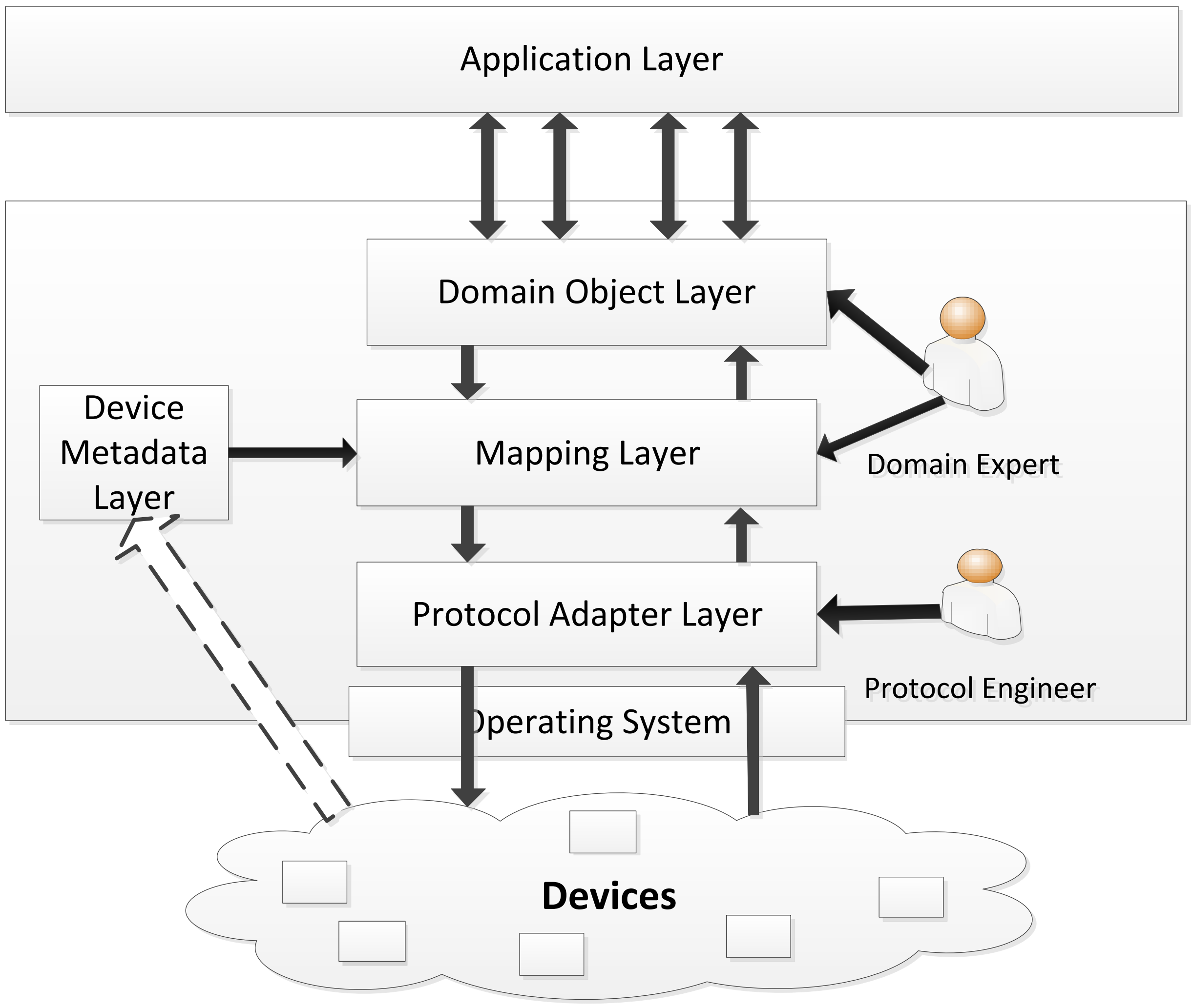


Figure 1.1: Block diagram of the Middleware architecture

Finally, the operating system provides run-time support to transmit the protocol packets over a network to the device and receive packets from the device. The protocol adapter layer processes the received packets, and the domain objects are updated with the knowledge of mapping information.

The benefits of the proposed middleware which address the above explained middleware challenges are as follows:

- Device heterogeneity can be isolated in the mapping layer by providing a high level mapping between the domain object and the device.

- Developed protocol adapters are reusable across different applications.
- Modifications to the domain objects (to include a new data point) can be made by the domain experts, and do not require changes to the protocol adapters.
- Domain experts can focus on the concepts that are relevant, and can ignore protocol and device communication implementation details.

1.3 Organization of the Thesis

The thesis is organized into five chapters as follows:

- **Chapter 2:** This chapter presents a review of existing approaches addressing different challenges in distributed application development.
- **Chapter 3:** This chapter describes abstract protocol adapter requirements and the middleware's key design concepts followed by a description of the high level design model of the middleware framework that includes the class model and sequence diagrams.
- **Chapter 4:** This chapter describes the case studies implementing the high-level design model illustrated in chapter 3. The case studies include a detailed description of the domain model and the communication protocol used for data exchange along with sample code snippets.
- **Chapter 5:** The final chapter highlights the important features of the proposed middleware and scope of future research.

Chapter 2

Literature Review

This chapter reviews the existing methods which address the application development challenges described earlier. The reviewed work mainly highlights middleware and model driven engineering approaches.

2.1 Middleware approaches

Middleware can be defined as software that acts as an intermediary between other software, applications, and/or devices [12]. Middleware was first introduced in the context of enterprise systems, in which it was used for integrating the disparate and heterogeneous computing systems across multiple units. It is also being used in the contexts of distributed systems, mobile computing and embedded systems [13].

Middleware offers common services for the applications and eases the application development process. Middleware for distributed systems facilitates development and deployment of many applications in the areas of industrial, environmental, medical and energy management domains. However, it presents a number of challenges [14, 15]. The challenges addressed in this study are device heterogeneity, software heterogeneity, abstract specification of the application data schema and re-usability of protocol adapters across many applications. A variety of sensor middleware that address these challenges are described in the literature. They can be classified into following three categories according to the nature of the deployed devices, application domain and the place where data query is processed.

- Wireless Sensor Network management
- Data Stream Management
- Hybrid Approaches

Middleware solutions in Wireless Sensor Networks (WSNs) usually act as controllers that provide functionality for programming global behavior in different sensor nodes [16]. The data query is processed in purely distributed manner. Milan (Middleware Linking Applications and Networks) [17] focuses on high-level concerns by providing an interface mainly characterized by applications that actively affect the entire network. Milan lets sensor network applications specify their quality needs and adjusts the network characteristics to increase application lifetime while still meeting those needs. Mires [18] proposes an adaptation of a message-oriented middleware for traditional fixed distributed systems. Mires provides an asynchronous communication model that is suitable for WSN applications, which are event driven in most cases. It adopts a component-based programming model using active messages to implement its publish-subscribe based communication infrastructure. However, in WSNs middleware solutions, the support for device and software heterogeneity is very limited because the deployed sensor nodes are homogeneous, and they implement the same communication and data exchange protocols. Since these solutions require a version of the middleware to run on the device itself, there is no support for COTS devices.

Data Stream Management Systems (DSMS) middleware solutions such as Aurora [19] and STREAM [20] uses central servers to evaluate continuous queries on data streams issued by sensors. Queries are formulated by using declarative query languages like CQL [21]. DSMS solutions are ideal for applications in which data processing time is critical, for example, financial applications. However, device and software heterogeneity including support for COTS are not explored in DSMS solutions and the application data schema is supposed to be in a format comprehensible by the DSMS middleware.

Middleware solutions in Hybrid approaches aim to take advantage of WSNs and DSMS categories while supporting device heterogeneity. Sstreamware [22] provides a simple data schema that allows data representation of different types of sensors in a common generic way. It also handles device and software heterogeneity by proxies and adapters. Brito et al. [23] proposed a hybrid middleware that supports the integration of heterogeneous medical devices which are by default COTS devices. This middleware also provides support to run simple data pre-processing algorithms as services. It can be invoked from Matlab 7.x, which enables researchers to test the developed algorithms with the data acquired from any supported sensor in real-time. However, the authors don't mention any reference as to how a domain user can specify the application data schema in an abstract manner. Another hybrid middleware, GSN [24], is conceived for rapid and simple deployment of applications that need integration of data from heterogeneous sensors. The idea of 'Virtual Sensors' was introduced to abstract the sensors from the physical implementations and provide a homogeneous view of sensor data. A 'Virtual sensor' description includes a description of the application data schema and wrapper/adapter information. However, in GSN based applications, support for COTS devices is not discussed.

NCS is a system which consists of entities such as sensors, actuators, and control systems distributed over a network, and their corresponding control-loops are formed through a network layer. Kim et al. [25] discuss the general features and challenging issues of NCSs and propose a value of design and implementation of a middleware as a solution to address the challenges of NCSs. The value of abstractions, architecture, middleware, services in NCSs are discussed by Baliga et al. [26] and a message-oriented middleware called 'Etherware' was proposed. Etherware provides interfaces for the creation, upgrade, and migration of components. It allows local/remote components to communicate through compatible protocols. The architecture of Etherware is based on the micro kernel concept [27]. The *Kernel* represents the system invariant whose only function is to manage components and deliver messages between them. However, in Etherware, the remote components communicate via fixed set of protocols that limits the support for proprietary protocols and also there is limited or no support for COTS devices since the remote devices should be aware of Etherware.

2.2 Model Driven Engineering(MDE) approaches

Model Driven Engineering (MDE) technologies are used to raise the level of abstraction in application data schema specification and automate distributed application development process. MDE technologies use models at different levels of abstraction in application development process. An increase of automation in application development is achieved by using executable model transformations. Higher-level models are transformed into lower-level models until the model can be made executable, by using code generation or model interpretation tools. A model is often specified in some notation or language. Since modeling languages are mostly tailored to a particular domain, such language is often called a Domain-Specific Language (DSL).

Doddapaneni et al. [28] proposed an MDE framework for architecting and analyzing wireless sensor networks. This MDE framework proposes three modeling languages to model the software architecture of the WSN, the hardware specification of sensor nodes and the physical environment where the nodes are deployed. These models are linked together by using weaving models to generate the simulation scripts which are used to analyze the energy consumption of the modeled application. However, the software model components are physically deployed on hardware node components which limits the support for COTS devices and other proprietary protocols.

Di Marco et al. [29] proposed a UML based model driven approach that allows modeling of existing middleware Agilla agents [30] providing an easy way to design and implement a WSN application for non-expert programmers. This UML based approach also automatically generates middleware agents code from UML models using Model-to-Code transformations, where the generated code is adapted without service interruptions through agent substitutions. This study focus on providing an abstraction for non-expert programmers for an existing middleware approach, where device and software heterogeneity is addressed by underlying middleware.

Fleurey et al. [31] proposed a Domain-Specific Modeling Language called ThingML to support efficient development of IoT applications over heterogeneous WSNs. ThingML aims at promoting software engineering best practices for the specific case of resource-constrained systems. Application program interfaces (APIs) for serialization and deserialization of the communication messages between the devices are fully generated from the ThingML description by underlying model transformations. Even interactive simulators are generated to enable fast and early testing of the software development cycle. However, the communication protocol is fixed between the devices and the support for COTS devices is not addressed in the paper.

2.3 Summary

The literature review describes the existing methods such as middlware and MDE to address the application development challenges for distributed systems. Advantages and limitations of different categories (WSNs, DSMS, Hybrid) of middleware and MDE solutions are discussed. The proposed mapping technique between application data schema and device meta-data is implemented as middleware framework is explained in the next chapter.

Chapter 3

Middleware High level Design Model

This chapter describes the key concepts underlying the proposed middleware design, which is followed by the description of the middleware's high-level design model (that includes class model and sequence diagrams).

3.1 Key concepts

The middleware design model is based on the following key concepts.

3.1.1 Key Protocol Concepts

For this work, some popular communication protocols (Modbus, DNP3) were analyzed, and the following common protocol concepts were identified.

3.1.1.1 Packet Frame:

A packet frame is the structure of data in a protocol packet. The structure of a packet depends on the protocol packet type. A packet typically has a header and a payload. The header maintains transmission-related details and the overhead information about the packet. The payload is the data it carries. For example, Figure 3.1 shows an Ethernet packet frame.

3.1.1.2 Data Format:

Data format determines the manner in which the data in a protocol packet is interpreted. For example, the first 4 bytes in a protocol packet are interpreted as a UINT32 (un-signed 32-bit integer) number, and the packet is encoded in Big Endian format, where the most significant bytes of the data is stored at the lowest storage address.

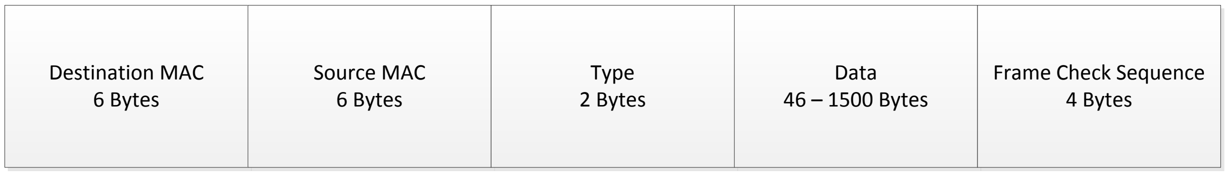


Figure 3.1: Ethernet Packet Frame

3.1.1.3 Object Identity:

The main purpose of a communication protocol is to transfer data between two devices. The device should be accessible and addressable in-order to communicate via the protocol. Every device/object in a network that uses a communication protocol has a unique identity for the purpose of identification and addressing.

3.1.1.4 Transaction Management:

A transaction is a sequence of information exchanged through the protocol packets between two computing entities. A transaction identifier which can be a 'Unique Sequence Number' is used to keep track of the protocol packet in a specific transaction. Each transaction in a communication serves a particular purpose, which is identified by the transaction type. Based on the semantics of transactions, they can be categorized into different transactional models. In the context of communication between the application and a device, we have 'Request-Reply' and 'Push' transactional models. This work focuses on 'Request-Reply' model.

Request-Reply vs. Push transactional models: A Request-Reply transactional model has two steps. Firstly, the application sends a request to the device and awaits a response, and then the device sends a reply in response. Usually there is a series of such interchanges until the complete message is sent. In Push transactional model, a transaction is issued without its corresponding request. A push transaction without a request is a unidirectional transaction which is useful for sending updates, readings and notifications of error conditions periodically from the device.

3.1.2 Key Middleware concepts

The following are key middleware concepts that underlie the approach described in this thesis.

3.1.2.1 Domain Object:

A domain object is an encapsulation of the application data schema that is read from a device or written to the device by a domain expert.

3.1.2.2 Device Metadata:

Device metadata describes the meta-information about the data provided by a particular device. The meta-information includes the type and the encoding format of the data, address location (memory map) of the data or the byte offset of the data in the protocol packet, and the description of the data provided by the device. This information also acts as data formatting rules for supported communication protocols.

3.1.2.3 Domain Map:

A domain map specifies the mapping between the domain object and device metadata. The mapping information is useful in translating the domain object into specific communication protocol packet and in updating the domain object from the protocol packet appropriately.

3.1.2.4 Protocol Adapter:

A protocol adapter provides interfaces to format the domain object into a protocol packet and update it from the packet.

3.1.2.5 End Point:

An end point provides appropriate interfaces for underlying network/serial communication to the physical device.

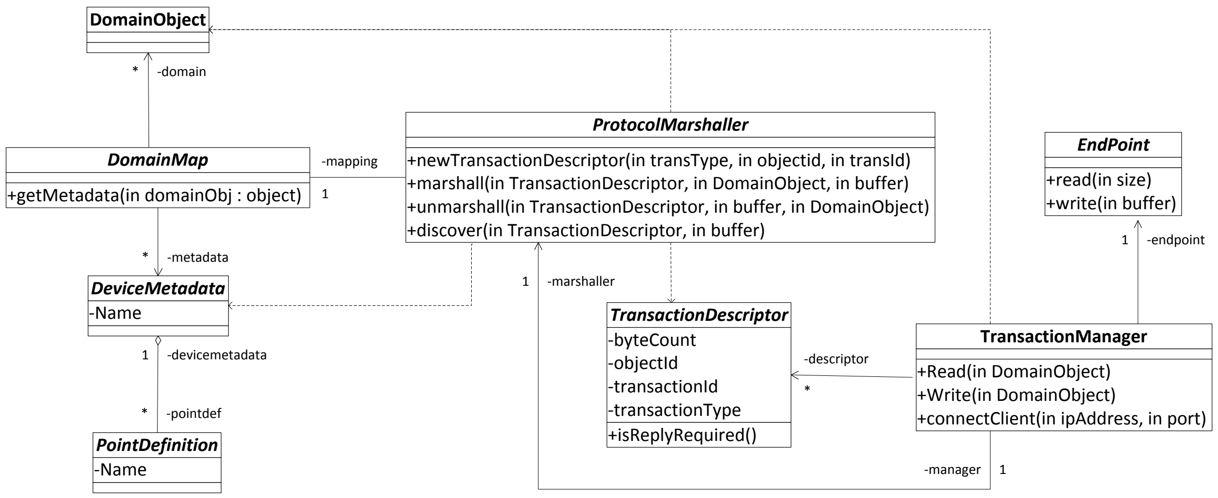


Figure 3.2: Midleware: High Level Class Model

3.2 Middleware High Level Design Model

In this section, the high-level design model of the middleware and the transaction management design issues are discussed. Figure 3.2 shows the high level design class model of the middleware. It describes the middleware's abstract entities and their corresponding interfaces. These entities must be specialized to provide core functionality for specific devices and their respective communication protocols.

The PointDefinition, DeviceMetadata, EndPoint, ProtocolMarshaller, DomainMap and TransactionDescriptor classes are abstract. Their specializations are responsible for the concrete device metadata and protocol implementations.

The PointDefinition abstract class is responsible for maintaining meta-information of a particular data point provided by the device, such as, type and format of the data point, data point address location, and the description of the data point provided by the device. The DeviceMetadata abstract class is composed of PointDefinitions and is responsible for providing an interface to access them. The EndPoint abstract class provides interfaces for different concrete network/serial endpoints which are responsible for opening a network stream or serial connection between the application and the physical device. The DomainObject class represents the encapsulation of the data that is read from an EndPoint or written to an EndPoint by the domain expert.

A protocol specific adapter is implemented by specializing the ProtocolMarshaller abstract class. This class is responsible for marshalling and unmarshalling of the domain object. It defines the 'marshall' and 'unmarshall' abstract operations, and the onus is on the Protocol-Marshaller's specialized classes to implement them. The 'marshall' operation transforms the domain object to a protocol specific packet, and the 'unmarshall' operation processes the packet data and updates the domain object. The 'marshall' and 'unmarshall' operations use the mapping information provided by the DomainMap class to transform the domain data to packet data and vice versa.

The DomainMap abstract class represents the logical mapping between domain objects and device metadata. A protocol specific mapping is implemented by specializing the abstract DomainMap class. The mapping information maps the domain objects and the device metadata that is helpful in formatting the domain object into protocol specific packet data.

The TransactionDescriptor class represents an abstract notion of a transaction and maintains important properties of a transaction such as its unique identifier, number of bytes of data in the transaction, transaction type (read-request/response and write-request/response) and device identity (object id). The TransactionDescriptor class can be specialized to maintain additional protocol specific properties.

The TransactionManager class is responsible for the transaction management process and it provides 'Read' and 'Write' operations. These operations are used by the domain expert to read or write the domain data to the device using specialized ProtocolMarshaller class as a protocol adapter. The 'Read' and 'Write' operations abstracts complete marshalling and unmarshalling of the domain object from a domain expert point of view. The interactions involved in 'Read'/'Write' operations are illustrated in the next subsection. The TransactionManager class is also responsible for creating and managing the transactions (also known as TransactionDescriptors) between the domain applications and the devices.

3.2.0.6 Transaction Management design concerns:

The important design concerns in managing transactions such as packet completeness and packet identification are described below:

Packet Completeness: During 'Read' or 'Write' operations, while the TransactionManager is waiting to process the response for the requested data, the TransactionManager queries the EndPoint interfaces for the response packet. The received packet might be a partial response packet in which case, the TransactionManger has to continue the querying process until the packet is complete enough for unmarshalling process.

Packet Identification: The packet identification process involves identifying the received packet type from an EndPoint i.e. whether the received packet is of response-read or response-write or incomplete transaction type. The knowledge of the packet type is required for unmarshalling process of the packet data.

TransactionManager addresses packet completeness and packet identification concerns by calling ProtocolMarshaller's 'discover' method. This method identifies the packet type based on the protocol knowledge by processing the received packet and updates the TransactionDescriptor's properties (like transacitonid, transactiontype and bytecount). For an 'incomplete' transactiontype, the packet data is not complete enough for unmarshalling process to update the domain object. Therefore, the TransactionManager continues to query the underneath EndPoint stream until the transactiontype is not 'incomplete'.

If many 'Read' or 'Write' queries are issued to the EndPoint, there is a need to identify and map many responses to its source requests. To handle this, the TransactionManager class maintains a hash map of the issued transactions with transactionid as the key and domain object as its value. Whenever, a response is received, and its packet completeness and packet identification processes are successfully completed, the TransactionManager queries the hash map based on the updated transactionid to get the source transaction domain object. The queried domain object is updated by the ProtocolMarshaller's 'unmarshall' method from the response packet data.

3.2.1 Sequence Diagram for TransactionManager's Read Operation

The TransactionManager's 'Read' and 'Write' operations are used by the domain experts to read the data from the device and to write the data to a device respectively. In this section, the interactions involved in 'Read' operation are illustrated. The interactions for the 'Write' operation are similar with the exception of a change in state/value of parameters.

3.2.1.1 Read operation in Synchronous transaction

Figure 3.3 illustrates the interactions involved in simple a case of 'Read' operation. The domain expert calls the 'Read' operation with the domain object as a parameter. The interactions involved in Fig 3.3 are described below (with particular sequence numbers as shown in the diagrams):

1 A TransactionDescriptor instance is created by calling the 'newTransactionDescriptor' method of ProtocolMarshaller class with transaction type as ReadRequest. The TransactionDescriptor instance is created to maintain transaction particular attributes as explained earlier. It is also used to pass the hints (protocol specific control information) during marshalling/unmarshalling process.

2 The ProtocolMarshaller's 'marshall' method translates the domain object into a protocol request packet.

3 The request buffer from the 'marshall' method is sent to the device over the network by calling the 'write' method of EndPoint class.

4 The 'read' method of the EndPoint class is queried for the response packet from the device.

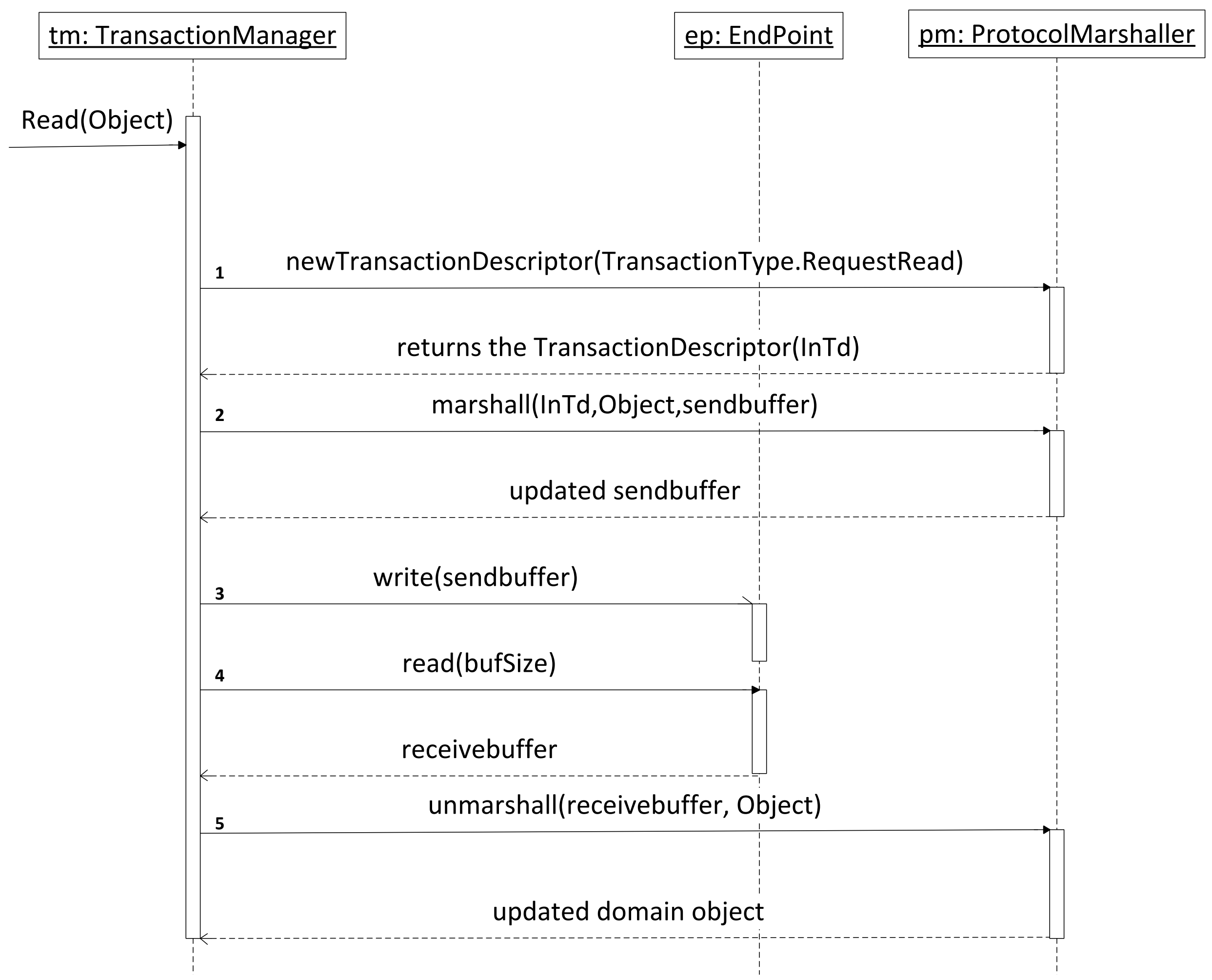

tm: TransactionManager
ep: EndPoint
pm: ProtocolMarshaller
Read(Object)
1
newTransactionDescriptor(TransactionType.RequestRead)
returns the TransactionDescriptor(InTd)
2
marshall(InTd,Object,sendbuffer)
updated sendbuffer
3
write(sendbuffer)
4
read(bufSize)
receivebuffer
5
unmarshall(receivebuffer, Object)
updated domain object

Figure 3.3: TransactionManager Read operation Sequence Diagram

5 The ProtocolMarshaller's 'unmarshall' method is invoked. The response packet is processed, and the domain object's data members are updated.

In reality, there is a need to address the transaction management implementation concerns such as packet identification, packet completeness and handling multiple 'Read' requests, as discussed earlier. Figure 3.4 illustrates the 'Read' operation interactions, which address these implementation concerns. The interactions involved are described below (with particular sequence numbers).

1 This interaction is similar to the first step in Fig 3.3. A new TransactionDescriptor's

instance is created by calling the 'newTransactionDescriptor' method of ProtocolMarshaller's class with transaction type as ReadRequest.

1a Update the TransactionDescriptor (InTD) instance data members such as transactionid, transactiontype and objectid.

1b Add the TransactionDescriptor's (InTD) transactionid as a key and domain object as a value to the transaction dictionary which is queried before 'unmarshall' method (interaction-9)

2 This interaction is similar to the second step as Fig 3.3, ProtocolMarshaller's 'marshall' method translates the domain object into a protocol request packet (buffer).

3 This interaction is is related to the third step in Fig 3.3, the updated request packet (buffer) is sent to the device over the network by calling the 'write' method of EndPoint class.

3a A new TransactionDescriptor instance is created by calling the 'newTransactionDescriptor' method of ProtocolMarshaller's class. This transaction descriptor is used for the packet identification process.

4 This interaction is similar to the fourth step in Fig 3.3 where, the 'read' method of the EndPoint class is queried for the response packet from the device.

4a The ProtocolMarshaller's 'discover' method is invoked to determine the completeness of the response packet. Step 4 and step 4a are run in a loop until the TransactionDescriptor's (OutTd) transactiontype property is not 'incomplete'.

5 The transaction dictionary is queried for the domain object based on the TransactionDescriptor's (OutTd) transactionid data member. The response packet is processed, and the domain object data members are updated by 'unmarshall' method of the concrete ProtocolMarshaller's class.

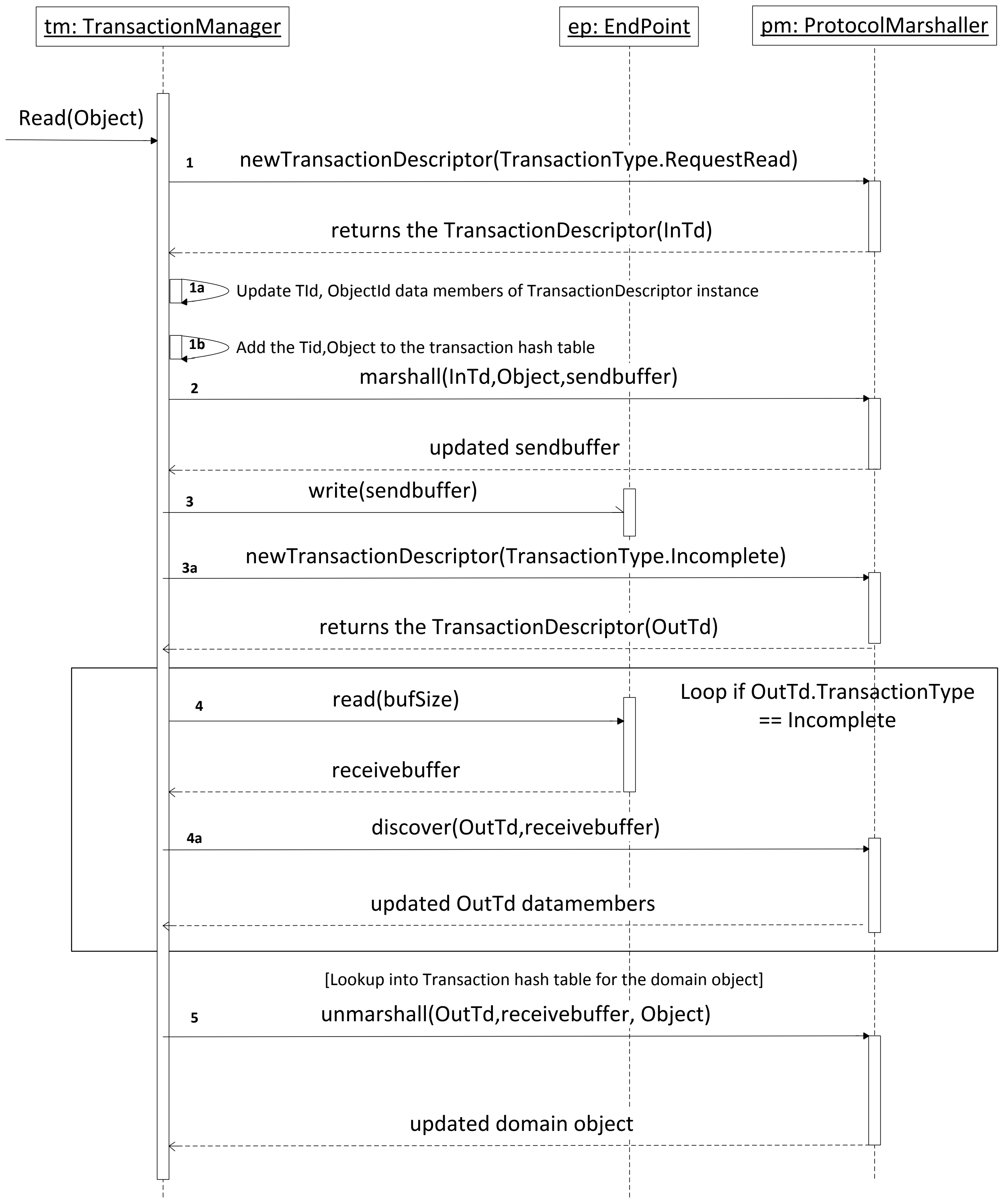

tm: TransactionManager
ep: EndPoint
pm: ProtocolMarshaller
Read(Object)
1
newTransactionDescriptor(TransactionType.RequestRead)
returns the TransactionDescriptor(InTd)
1a
Update TId, ObjectId data members of TransactionDescriptor instance
1b
Add the Tid,Object to the transaction hash table
2
marshall(InTd,Object,sendbuffer)
updated sendbuffer
3
write(sendbuffer)
3a
newTransactionDescriptor(TransactionType.Incomplete)
returns the TransactionDescriptor(OutTd)
Loop if OutTd.TransactionType == Incomplete
4
read(bufSize)
receivebuffer
4a
discover(OutTd,receivebuffer)
updated OutTd datamembers
[Lookup into Transaction hash table for the domain object]
5
unmarshall(OutTd,receivebuffer, Object)
updated domain object

Figure 3.4: TransactionManager Read operation Sequence Diagram

Chapter 4

Case Studies

This chapter describes the case studies implementing the high-level design model illustrated in chapter 3. The case studies include a description of the domain, the communication protocol, detail-level design class model of the middleware for Modbus enabled devices and the code samples.

4.1 First Case Study: Software Client in a NCS for Simulated Power System

The middleware, which is proposed and implemented in this study has already been successfully adapted as a software layer for multiple client communications in [1].

4.1.1 Domain Background

Jain [1], proposed and implemented a distributed network control framework (illustrated in Figure 4.1) for a power system simulation. The power system (or micro-grid) simulation runs on an Opal-RT real time simulation system [32] (acting as a server) with its devices including a photo-voltaic (PV) system and over-current (OC) relays communicates with the distributed clients for control data. One such client is termed as a 'software controller', which sends the control data (duty cycle) by computing Maximum Power Point Tracking (MPPT) algebraic algorithms to the inverter of a PV system in a feedback loop [33]. The MPPT controller is used to derive the maximum possible power from semiconductor devices like a photo-voltaic panel, in which the current does not vary linearly with voltage. The MPPT controller can use different algorithms to optimize the power output of a PV system.

The algorithm used in the software controller is most widely used algorithm *The Perturb & Observe Method (P&O)* [34]. The P&O method ensures maximum power output from

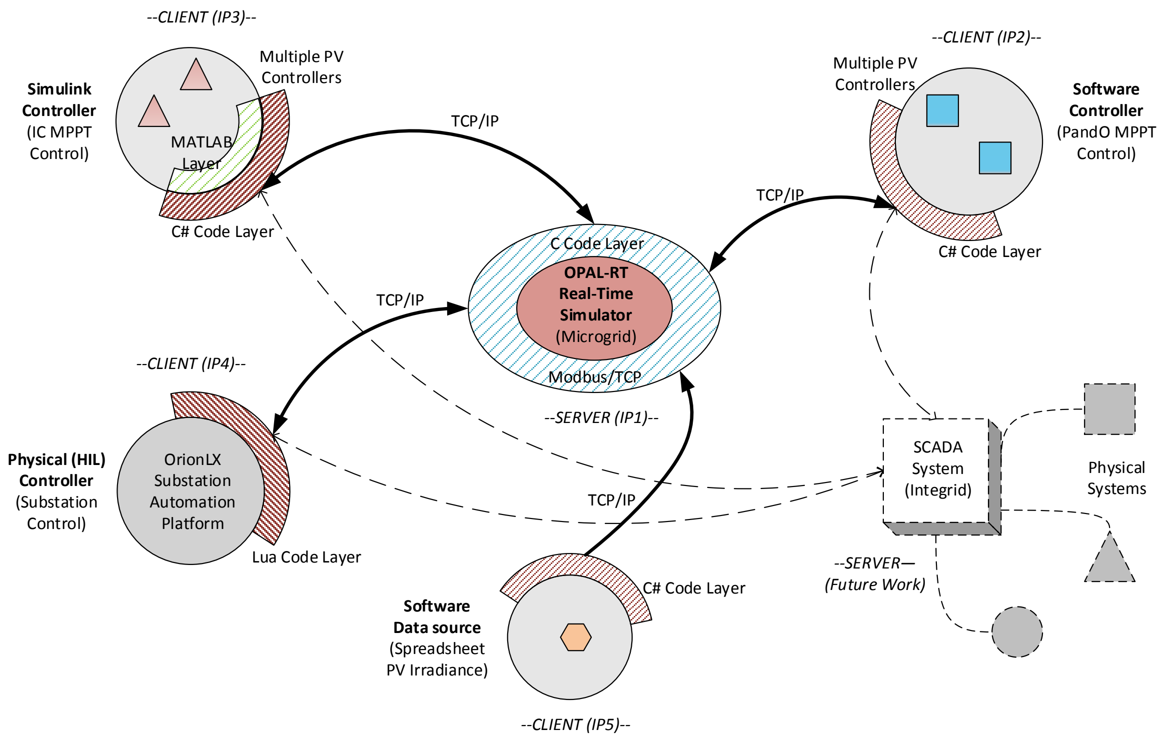


Figure 4.1: Distributed Networked Control Framework for a Microgrid [1]

the photo-voltaic panel by modifying its output operating voltage or current. In a paper by Jain [1], the implemented P&O algorithm takes voltage, current and time as input and generates duty cycle control data as output. The voltage and current input parameters are used to compute the control data (duty cycle), while the input parameter time is used for time-synchronization of various clients. In the context of this work, the domain data used are voltage, current, time and duty cycle. The domain data is processed by application level algorithm (P&O), and the data is communicated to the Opal-RT server (PV simulation). Modbus TCP/IP is used as the communication protocol to exchange the data between the application and the server. The Opal-RT server has a layer of C code that contains socket programming, as well as Modbus/TCP implementation.

4.1.2 Modbus Protocol

Modbus is an application layer communication protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs) and it has since become popular with industrial automation devices. Modbus allows for client-server communication between several devices connected over the network. There are different versions of this protocol [6]. The version implemented in this thesis is Modbus TCP/IP, which uses request-response transactional model for message passing mechanism between the application and the server.

4.1.2.1 Modbus Frame

Figure 4.2 illustrates the Modbus request/response frame format over TCP/IP. A typical Modbus TCP packet frame is composed of protocol data unit (PDU) and application data unit (ADU). In general, an application layer communication protocol defines a simple PDU independent of the underlying communication layers. Subsequently, when the protocol is mapped to a specific network, some additional fields are introduced, which constitutes to the application data unit (ADU). All the fields in the Modbus frame are encoded in Big-Endian byte ordering notation.

A dedicated Modbus header is used over TCP/IP network communication to identify the Modbus ADU. It is called MBAP header (Modbus Application Header). The length of Modbus TCP/IP ADU = PDU (253 bytes) + MBAP (7 bytes) = 260 bytes. The MBAP header contains the following fields:

- Transaction Identifier: This field is used for Modbus request/response transaction pairing between client and server. The transaction identifier is assigned by the client initiating the request and is recopied by the server in its response packet.
- Protocol Identifier: This field is used for intra-system multiplexing. The Modbus protocol is identified by the value '0' and this field has been reserved for future protocol

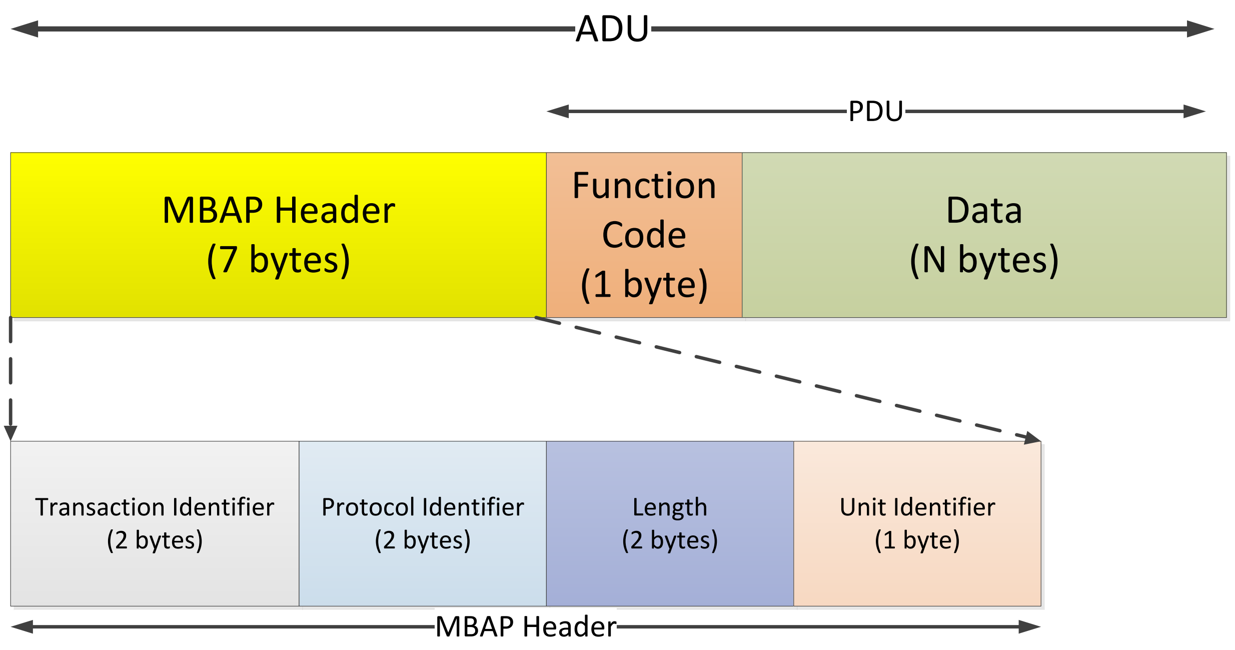


Figure 4.2: Modbus over TCP Frame Format

extensions. It is also assigned by the initiating client and is recopied by the server in response.

- Length: This field contains the byte count of 'Unit Identifier' and PDU fields. It is assigned separately by both the client and the server where they indicate their respective number of bytes.

- Unit Identifier: This field is used for intra-system routing. It is typically used to communicate to a Modbus serial line slave through a gateway between an Ethernet TCP/IP network and a Modbus serial line. This field is set by the Modbus client in the request and must be returned with the same value in the response by the server. Since the Unit ID is only 1 byte long, the maximum number of clients that can be connected to a single server, without modifying the protocol format is 255.

The Modbus PDU comprises of function code and data fields. Function code indicates the transaction type issued between the client and the server. The data field contains the information exchanged between the server and the client. The PDU fields are explained

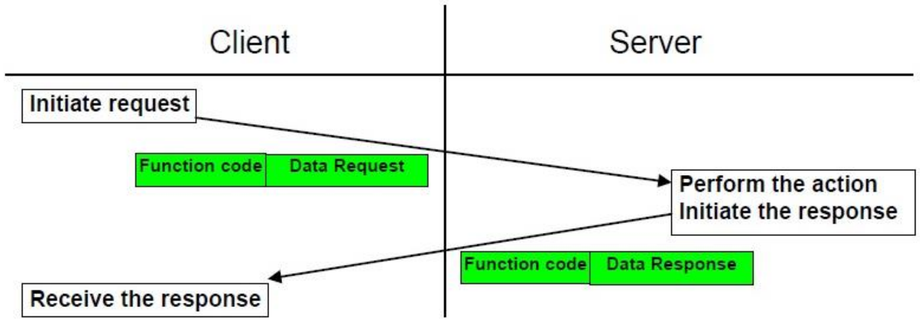


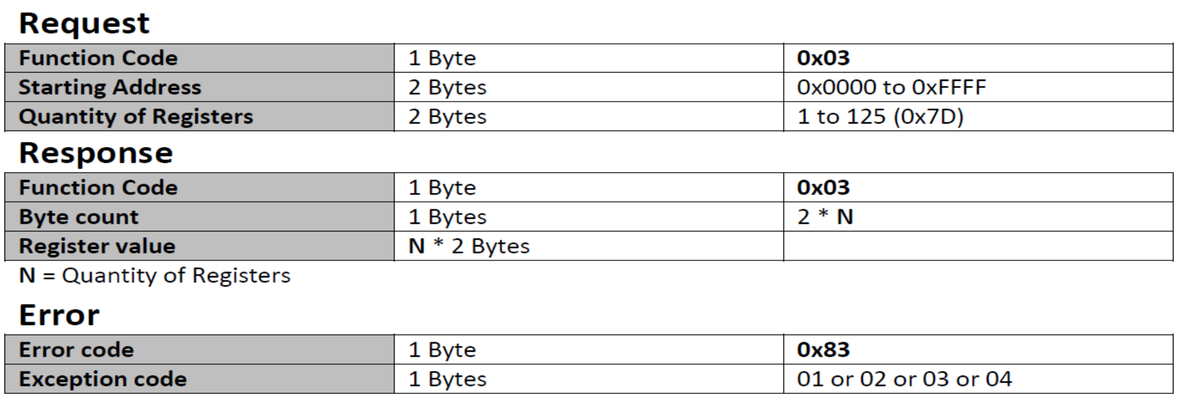
Figure 4.3: Modbus Transaction [2]

below [2]:

- Function Code: This code indicates to the server as to what function to perform with respect to a request from the client that is illustrated in Fig 4.3. With the size of 1 byte, function codes range from 1-255, where the codes 128-255 are reserved for exception handling. This work is focused on few of the function codes explained below:

  1. 03: Read Holding Registers: This function code is used by the client to indicate that it intends to read the contents of a contiguous block of holding registers from the indicated device.
  2. 06: Write Single Register: This function code is used to write a single holding register in the remote device.
  3. 16: Write Multiple Registers: This function code is used to write to a block of contiguous holding registers in the remote device.

- Data: The data field of the PDU contains additional information that the server uses to take the action defined by the function code. The content in this field varies based on the transaction type as shown in Fig 4.4 and Fig 4.5, where the format of read request packet and read response packet, write request packet and write response packet are different from each other respectively.



Request

Function Code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

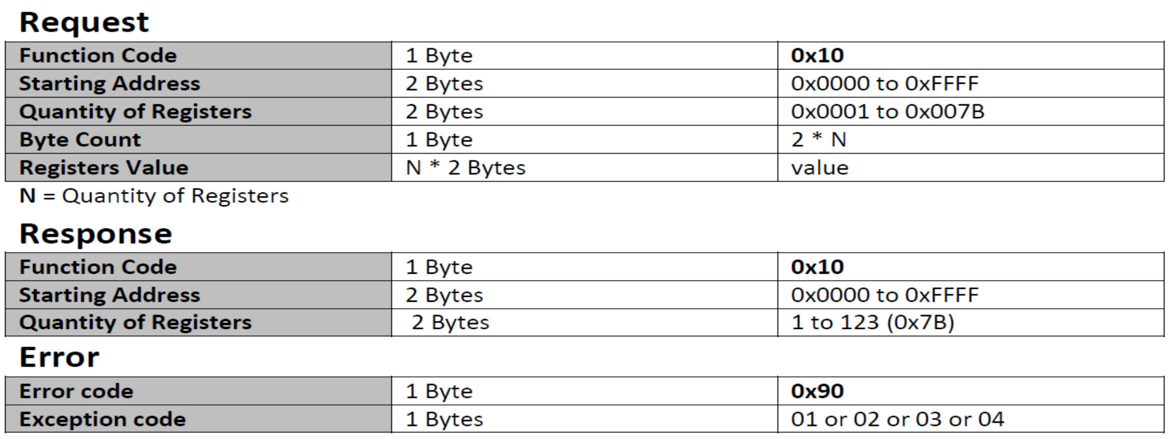
Function Code	1 Byte	**0x03**
Byte count	1 Bytes	2 * N
Register value	N * 2 Bytes	

N = Quantity of Registers

Error

Error code	1 Byte	**0x83**
Exception code	1 Bytes	01 or 02 or 03 or 04

Figure 4.4: Modbus Read request/response packet format [2]



Request

Function Code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2 * N
Registers Value	N * 2 Bytes	value

N = Quantity of Registers

Response

Function Code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 123 (0x7B)

Error

Error code	1 Byte	**0x90**
Exception code	1 Bytes	01 or 02 or 03 or 04

Figure 4.5: Modbus Write request/response packet format [2]

4.1.2.2 Modbus Data Model:

The data model is based on a series of tables, which include discrete input, coils, input registers and holding registers. These tables are word addressable data items, and they do not imply any application behavior. The protocol allows selection of 65536 data items for each of the tables and the read/write operations to these tables are dependent on the function code as stated earlier. The mapping between the Modbus data model and the device application is totally vendor specific, which adds a layer of complexity to develop a generic Modbus protocol marshaller for all Modbus devices. Since the registers are 16 bit (word) addressable units, the data can be spanned across the registers. The 32 bit value

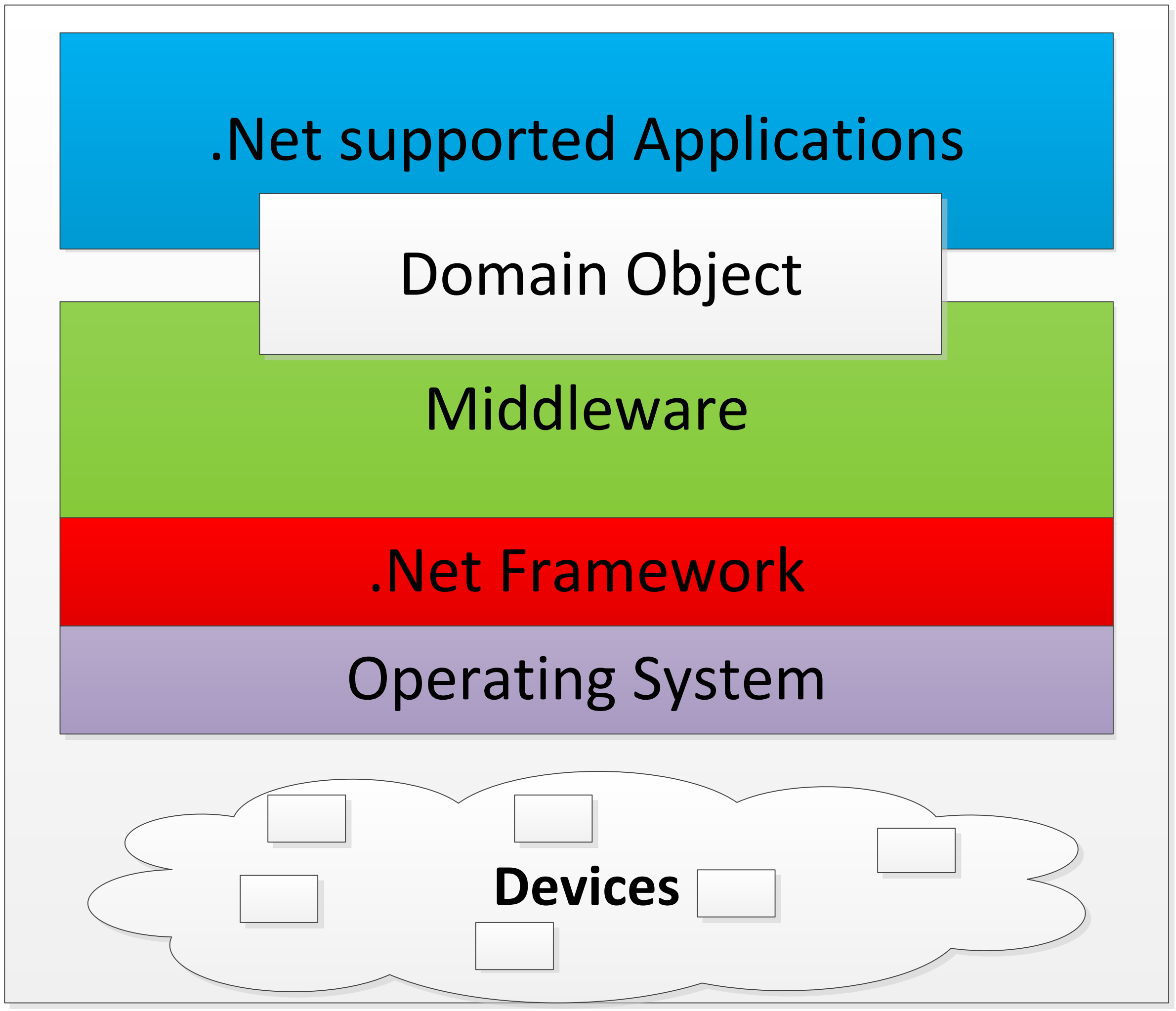


Figure 4.6: High level view of the Middleware Application Architecture

is spanned across two registers, and it can be represented as an integer or floating point number. Encoding these values into the register map is vendor specific. In most cases, the vendor provides a description of protocol-device (registers/coils) mapping, which describes the attributes addressed by device metadata.

4.1.3 Middleware-Detail Design class model for Modbus enabled devices

The middleware is implemented using .NET 4.5 framework [35] and C# programming language. A high-level view of the application architecture is illustrated in Fig 4.6, where the

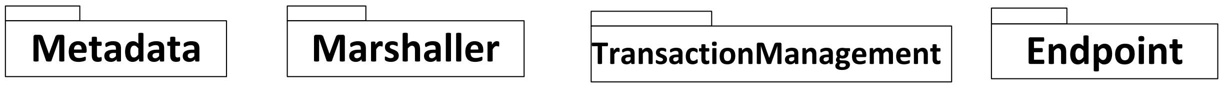


Figure 4.7: Middleware Detail Level Design Package Abstraction

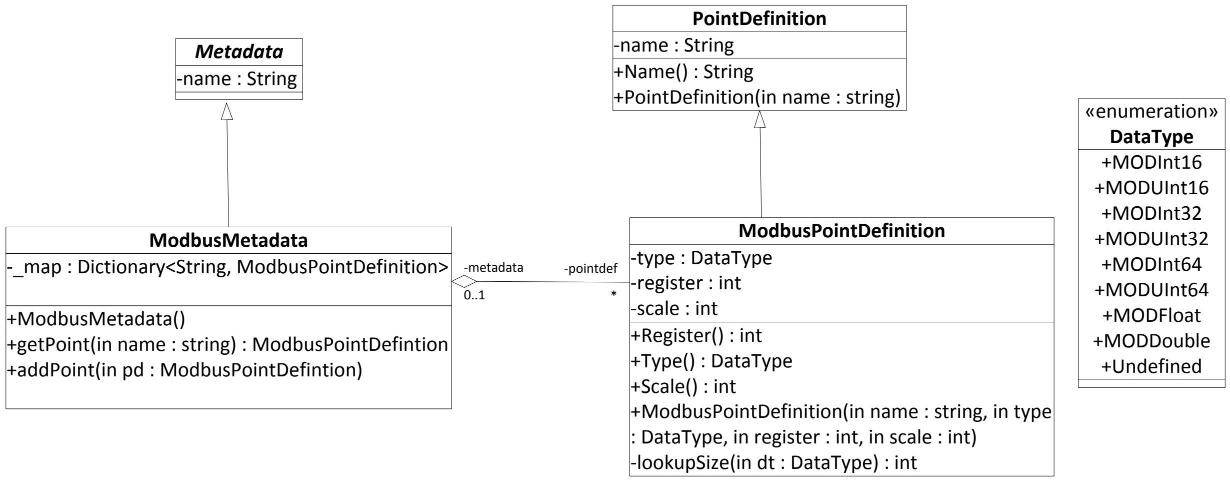
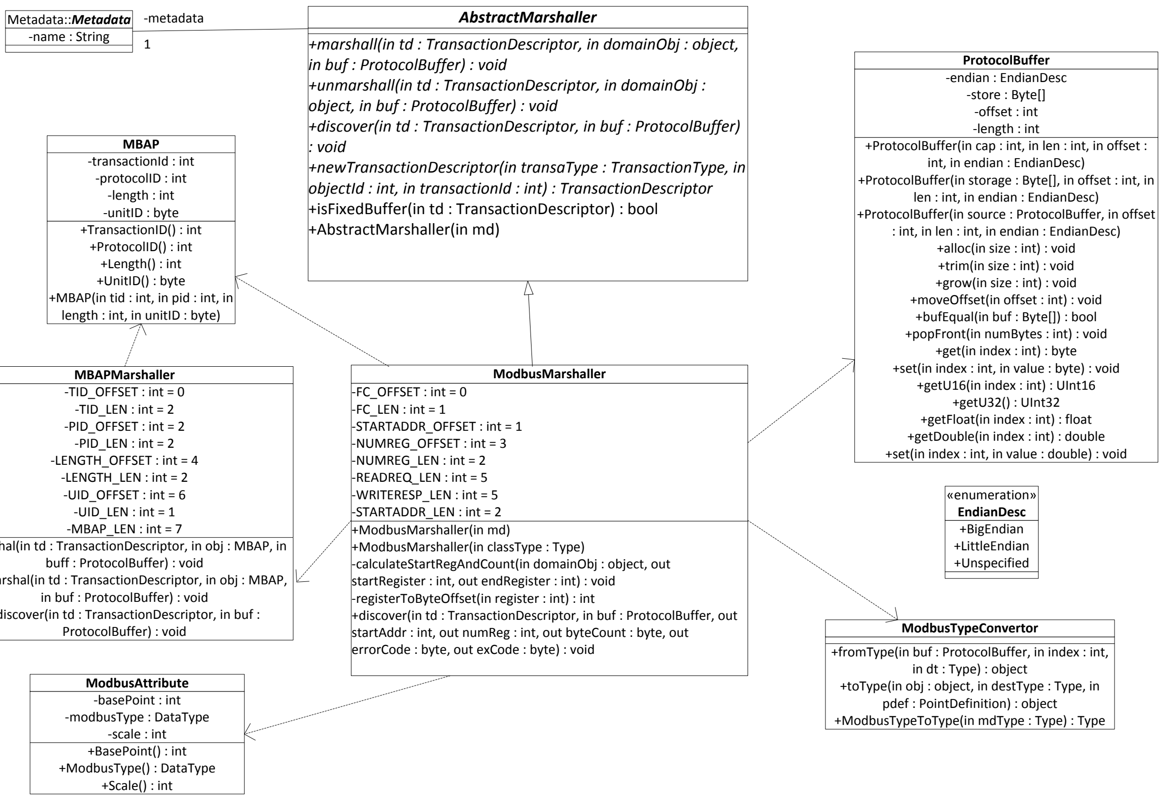


Figure 4.8: Metadata package

domain objects in the applications are defined in .NET compatible languages. The middleware's high-level design class model (Fig 3.2) explained in the previous chapter is specialized for Modbus devices. Figure 4.7 represents the detail-level design package abstraction which contains the Metadata, Marshaller, TransactionManagement and Endpoint packages. Figure 4.8 (Metadata Package), Fig 4.9 (Marshaller Package), Fig 4.10 (TransactionManagement Package) and Fig 4.11 (Endpoint Package) represents the concrete classes and their associations and dependencies among their respective packages. The device metadata classes for Modbus protocol supported devices are specialized in metadata package illustrated in Fig 4.8. Modbus protocol specific marshaller classes are illustrated in Fig 4.9. The TransactionManager and concrete endpoint classes are illustrated in Fig 4.10 and Fig 4.11.



Metadata::*Metadata*
-name : String
-metadata
1
AbstractMarshaller
+marshall(in td : TransactionDescriptor, in domainObj : object, in buf : ProtocolBuffer) : void
+unmarshall(in td : TransactionDescriptor, in domainObj : object, in buf : ProtocolBuffer) : void
+discover(in td : TransactionDescriptor, in buf : ProtocolBuffer) : void
+newTransactionDescriptor(in transaType : TransactionType, in objectId : int, in transactionId : int) : TransactionDescriptor
+isFixedBuffer(in td : TransactionDescriptor) : bool
+AbstractMarshaller(in md)
ProtocolBuffer
-endian : EndianDesc
-store : Byte[]
-offset : int
-length : int
+ProtocolBuffer(in cap : int, in len : int, in offset : int, in endian : EndianDesc)
+ProtocolBuffer(in storage : Byte[], in offset : int, in len : int, in endian : EndianDesc)
+ProtocolBuffer(in source : ProtocolBuffer, in offset : int, in len : int, in endian : EndianDesc)
+alloc(in size : int) : void
+trim(in size : int) : void
+grow(in size : int) : void
+moveOffset(in offset : int) : void
+bufEqual(in buf : Byte[]) : bool
+popFront(in numBytes : int) : void
+get(in index : int) : byte
+set(in index : int, in value : byte) : void
+getU16(in index : int) : UInt16
+getU32() : UInt32
+getFloat(in index : int) : float
+getDouble(in index : int) : double
+set(in index : int, in value : double) : void
MBAP
-transactionId : int
-protocolID : int
-length : int
-unitID : byte
+TransactionID() : int
+ProtocolID() : int
+Length() : int
+UnitID() : byte
+MBAP(in tid : int, in pid : int, in length : int, in unitID : byte)
MBAPMarshaller
-TID_OFFSET : int = 0
-TID_LEN : int = 2
-PID_OFFSET : int = 2
-PID_LEN : int = 2
-LENGTH_OFFSET : int = 4
-LENGTH_LEN : int = 2
-UID_OFFSET : int = 6
-UID_LEN : int = 1
-MBAP_LEN : int = 7
+marshal(in td : TransactionDescriptor, in obj : MBAP, in buff : ProtocolBuffer) : void
+unmarshal(in td : TransactionDescriptor, in obj : MBAP, in buf : ProtocolBuffer) : void
+discover(in td : TransactionDescriptor, in buf : ProtocolBuffer) : void
ModbusMarshaller
-FC_OFFSET : int = 0
-FC_LEN : int = 1
-STARTADDR_OFFSET : int = 1
-NUMREG_OFFSET : int = 3
-NUMREG_LEN : int = 2
-READREQ_LEN : int = 5
-WRITERESP_LEN : int = 5
-STARTADDR_LEN : int = 2
+ModbusMarshaller(in md)
+ModbusMarshaller(in classType : Type)
-calculateStartRegAndCount(in domainObj : object, out startRegister : int, out endRegister : int) : void
-registerToByteOffset(in register : int) : int
+discover(in td : TransactionDescriptor, in buf : ProtocolBuffer, out startAddr : int, out numReg : int, out byteCount : byte, out errorCode : byte, out exCode : byte) : void
«enumeration»
EndianDesc
+BigEndian
+LittleEndian
+Unspecified
ModbusTypeConvertor
+fromType(in buf : ProtocolBuffer, in index : int, in dt : Type) : object
+toType(in obj : object, in destType : Type, in pdef : PointDefinition) : object
+ModbusTypeToType(in mdType : Type) : Type
ModbusAttribute
-basePoint : int
-modbusType : DataType
-scale : int
+BasePoint() : int
+ModbusType() : DataType
+Scale() : int

Figure 4.9: Marshaller package

4.1.3.1 Domain object Mapping and device metadata specification

The mapping [1] between the domain object and device metadata is achieved by .NET *reflection* and the device metadata is specified as C# *attribute* features. The .NET framework's reflection API provides interfaces to access an object's attributes, properties and methods at run-time. An attribute is an object that represents data which can be associated to an element in the program.

In this work, the device metadata is specified in an implicit manner by decorating the domain object data members by the C# attributes. For example, ModbusAttribute class illustrated in Fig 4.9 is decorated around the domain object data members which are pro-

[1]Note: In the current version of the middleware, the mapping is implemented in an implicit manner rather by specializing the DomainMap class.

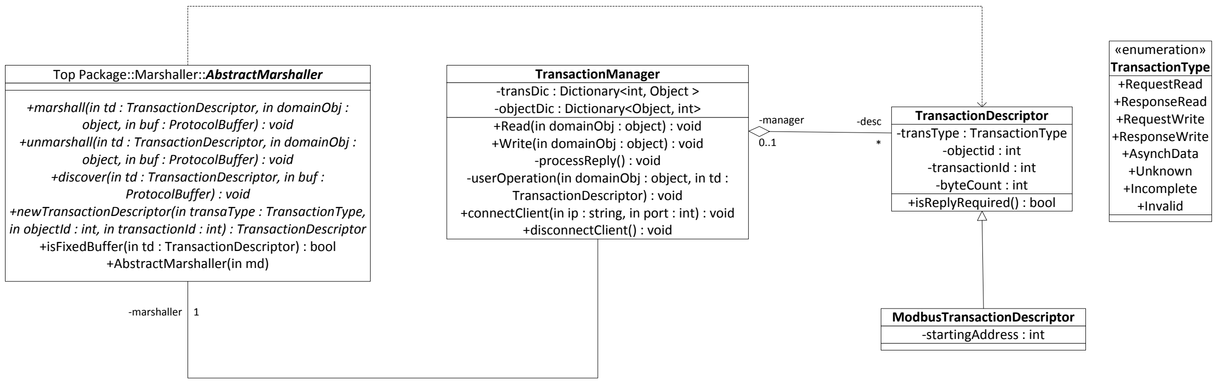

Top Package::Marshaller::AbstractMarshaller
+marshall(in td : TransactionDescriptor, in domainObj : object, in buf : ProtocolBuffer) : void
+unmarshall(in td : TransactionDescriptor, in domainObj : object, in buf : ProtocolBuffer) : void
+discover(in td : TransactionDescriptor, in buf : ProtocolBuffer) : void
+newTransactionDescriptor(in transaType : TransactionType, in objectId : int, in transactionId : int) : TransactionDescriptor
+isFixedBuffer(in td : TransactionDescriptor) : bool
+AbstractMarshaller(in md)
-marshaller
1
TransactionManager
-transDic : Dictionary<int, Object >
-objectDic : Dictionary<Object, int>
+Read(in domainObj : object) : void
+Write(in domainObj : object) : void
-processReply() : void
-userOperation(in domainObj : object, in td : TransactionDescriptor) : void
+connectClient(in ip : string, in port : int) : void
+disconnectClient() : void
-manager
0..1
-desc
*
TransactionDescriptor
-transType : TransactionType
-objectid : int
-transactionId : int
-byteCount : int
+isReplyRequired() : bool
ModbusTransactionDescriptor
-startingAddress : int
«enumeration»
TransactionType
+RequestRead
+ResponseRead
+RequestWrite
+ResponseWrite
+AsynchData
+Unknown
+Incomplete
+Invalid

Figure 4.10: Transaction Management package

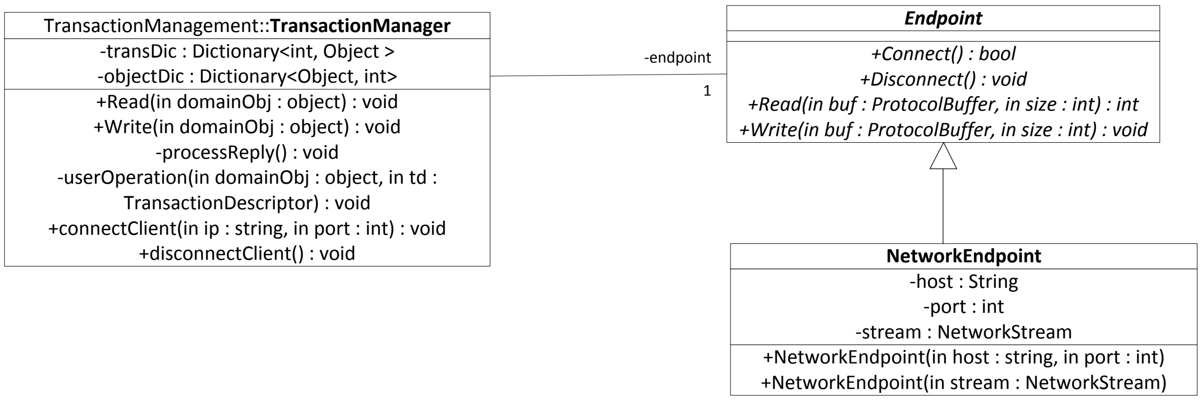

TransactionManagement::TransactionManager
-transDic : Dictionary<int, Object >
-objectDic : Dictionary<Object, int>
+Read(in domainObj : object) : void
+Write(in domainObj : object) : void
-processReply() : void
-userOperation(in domainObj : object, in td : TransactionDescriptor) : void
+connectClient(in ip : string, in port : int) : void
+disconnectClient() : void
-endpoint
1
Endpoint
+Connect() : bool
+Disconnect() : void
+Read(in buf : ProtocolBuffer, in size : int) : int
+Write(in buf : ProtocolBuffer, in size : int) : void
NetworkEndpoint
-host : String
-port : int
-stream : NetworkStream
+NetworkEndpoint(in host : string, in port : int)
+NetworkEndpoint(in stream : NetworkStream)

Figure 4.11: Endpoint package

cessed during run-time via reflection. The 'name' of the data member and the decorated attribute information represents device metadata information. The 'domain - protocol mapping' is achieved by querying the device metadata information (Basetype and ModbusType) based on the 'name' of the domain object's data member during run-time using reflection. Detailed code walk through of the domain objects, domain mapping, metadata, marshalling and unmarshalling process is described in next the subsection.

4.1.4 Case study - logical sample code walk through

This section describes the code snippets of the implemented middleware to read V, I and Time values from the PV simulation and to write the computed control data i.e. the duty cycle to the PV simulation.

4.1.4.1 Domain Objects

The InverterRead and InverterWrite classes encapsulate the data read from the PV simulation or written to the PV simulation by the control engineer. The InverterRead data members V, I and simTime represent the data to be read from the simulation. The P&O MPPT control algorithm as described earlier in the domain background section takes InverterRead data members as parameters and computes the InverterWrite class data member duty cycle (control data), which is then written to the PV simulation. The domain objects code snippet is illustrated in listing:4.1.

Code Snippet 4.1: Domain Objects

```
public class InverterRead
 {
     private double _V;
     private double _I;
     private double _sTime;
     public InverterRead(double V = 0.0,
     double I = 0.0, double simTime = 0.0)
     {
         _V = V;
         _I = I;
         _sTime = simTime;
     }
     [Modbus(BasePoint = 0,
      MODBUSType = ModbusPointDefinition.DataType.MODDouble)]
     public double V
     {
         get { return _V; }
         set { _V = value; }
     }
     [Modbus(BasePoint = 4,
     MODBUSType = ModbusPointDefinition.DataType.MODDouble)]
     public double I
     {
         get { return _I; }
         set { _I = value; }
     }
     [Modbus(BasePoint = 8,
     MODBUSType = ModbusPointDefinition.DataType.MODDouble)]
     public double simTime
     {
         get { return _sTime; }
         set { _sTime = value; }
     }
 }
 public class InverterWrite
 {
     private double _duty;
     public InverterWrite(double Start = 0.0)
     {
         _duty = Start;
     }
     [Modbus(BasePoint = 0,
      MODBUSType = ModbusPointDefinition.DataType.MODDouble)]
     public double Duty
     {
         get { return _duty; }
         set { _duty = value; }
     }
 }
```

4.1.4.2 Metadata specification and Mapping

It is evident from the domain objects code snippet illustrated in listing:4.1, that the data members are decorated with *Modbus attribute* which contains named parameters *BasePoint* and *MODBUSType*. The Modbus attributes and the *name* of the data member accounts for device metadata. For example, consider InverterRead class code snippet, the name of the data member *V* specifies the description of the data provided by the simulation i.e. V - voltage. MODBUSTYPE = MODDouble represents the data type and BasePoint = 0 represents the data address location (memory map) (which is the starting register address).

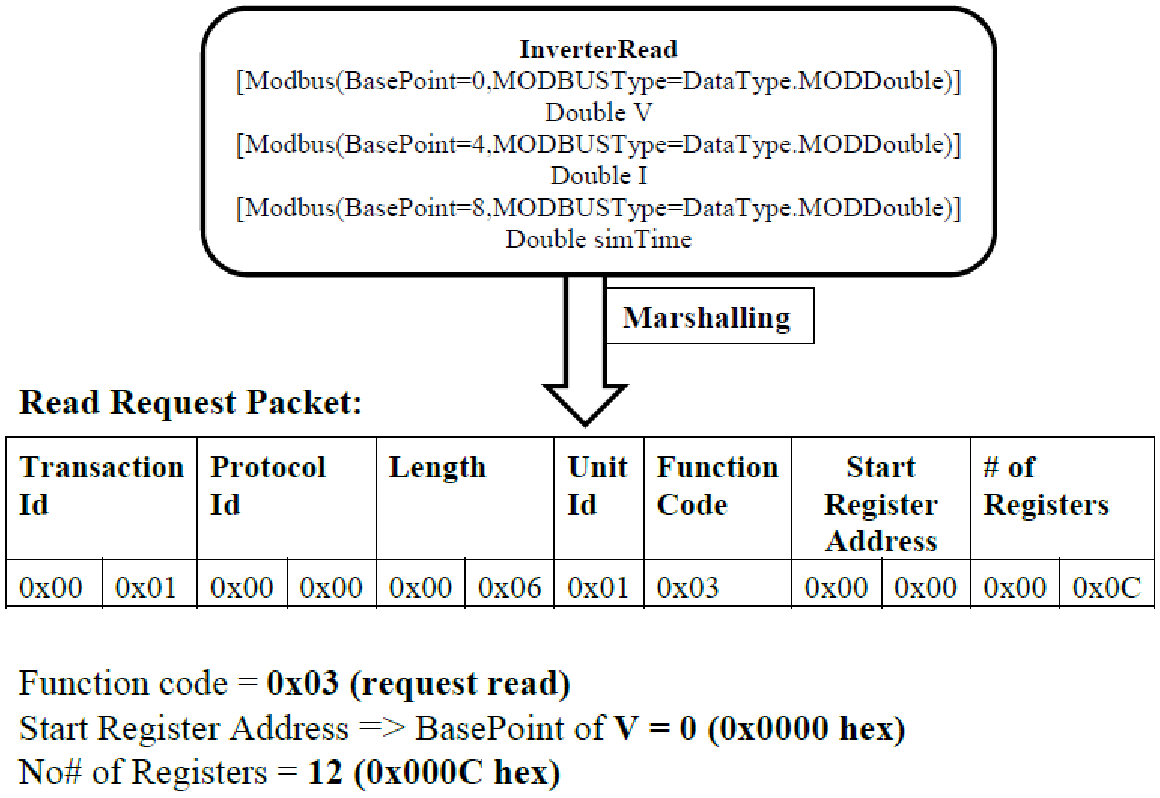
During run-time, using .NET reflection, these Modbus attributes of the InverterRead/InverterWrite object's data members are processed, and the ModbusPointDefinition's objects are created and added to the ModbusMetadata instance, the sample code snippet is illustrated in listing:4.2. Domain object-device metadata mapping for a specific data member is achieved by querying the ModbusPointDefinition's BasePoint and MODBUSType information based on the 'name' of the data member (Since the PointDefinition's 'name' data member is initialized from the domain object's data member 'name') during run-time using .NET reflection.

Code Snippet 4.2: Creation of ModbusPointDefintion via .NET reflection and C# attribute features

```
Type runtimeObj = inverterReadObj.GetType();
PropertyInfo[] arrProperties = runtimeObj.GetProperties();
foreach (PropertyInfo pi in arrProperties)
{
        object[] attributes = pi.GetCustomAttributes(
        typeof(ModbusAttribute),false);
    foreach (Object attribute in attributes)
    {
        ModbusAttribute ma = (ModbusAttribute)attribute;
        ModbusPointDefinition mpd = new ModbusPointDefinition(pi.Name,
        ma.ModbusType, ma.BasePoint);
        ModbusMetadata.addpoint(mpd)
    }
}
```

4.1.4.3 Marshalling and Unmarshalling of Domain Objects

The interactions involved in reading the domain data from the PV simulation or writing the domain data to the PV simulation via TransactionManager's Read/Write operations are illustrated in Fig 3.4. In this subsection, an overview of marshalling and unmarshalling


InverterRead
[Modbus(BasePoint=0,MODBUSType=DataType.MODDouble)]
Double V
[Modbus(BasePoint=4,MODBUSType=DataType.MODDouble)]
Double I
[Modbus(BasePoint=8,MODBUSType=DataType.MODDouble)]
Double simTime

Read Request Packet:

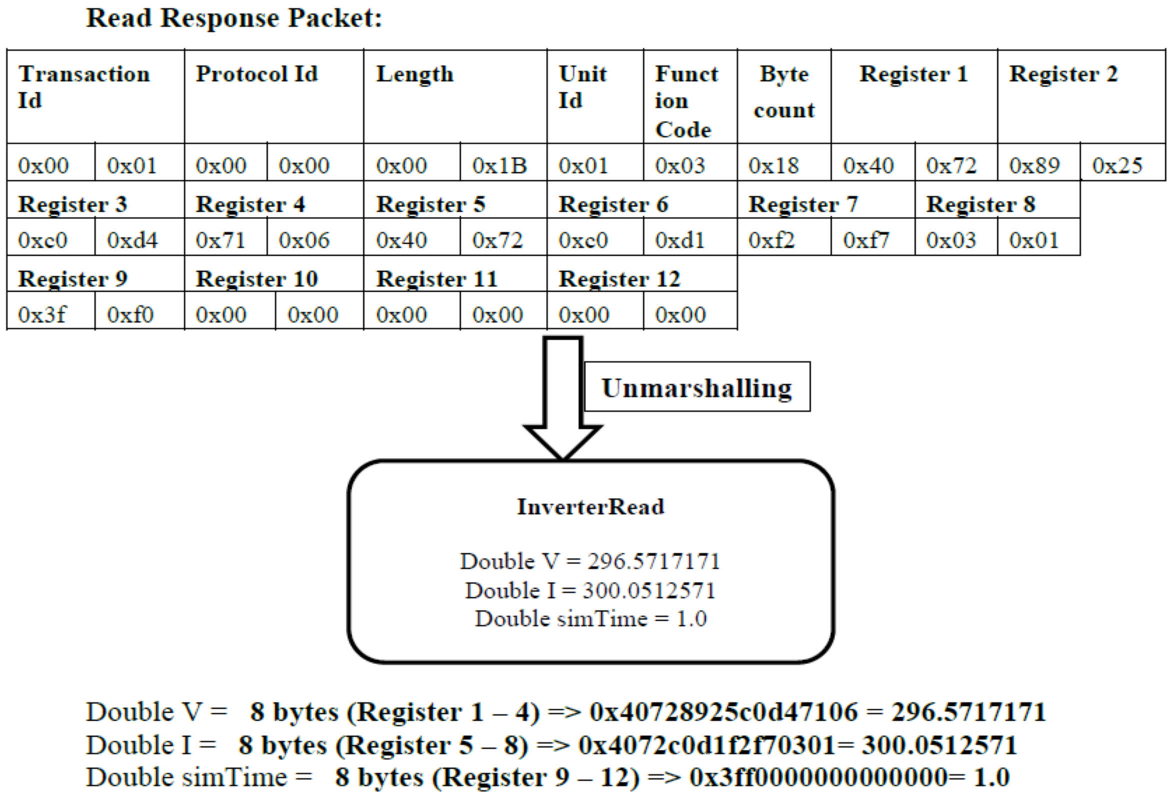
Transaction Id		Protocol Id		Length		Unit Id	Function Code	Start Register Address		# of Registers	
0x00	0x01	0x00	0x00	0x00	0x06	0x01	0x03	0x00	0x00	0x00	0x0C

Function code = **0x03 (request read)**
Start Register Address => BasePoint of **V = 0 (0x0000 hex)**
No# of Registers = **12 (0x000C hex)**

Figure 4.12: Marshalling output of the InverterRead Object

process of InverterRead and InverterWrite objects and their corresponding protocol buffer structure is discussed.

Marshalling of InverterRead object is illustrated in Fig 4.12, by invoking ModbusMarshaller's marshall method. It takes three parameters, firstly, an instance of TransactionDescriptor with transactiontype as RequestRead since a request to read specific values is made, secondly, the InverterRead domain object and thirdly, an instance of ProtocolBuffer. Based on the knowledge of the passed in parameters and its ModbusMetadata reference, the marshall method queries the domain object's mapping information for the byte offset (basepoint) and the data type (ModbusType). The mapping information is used to update the instance of the ProtocolBuffer to a well formed Modbus request read packet corresponding



Read Response Packet:

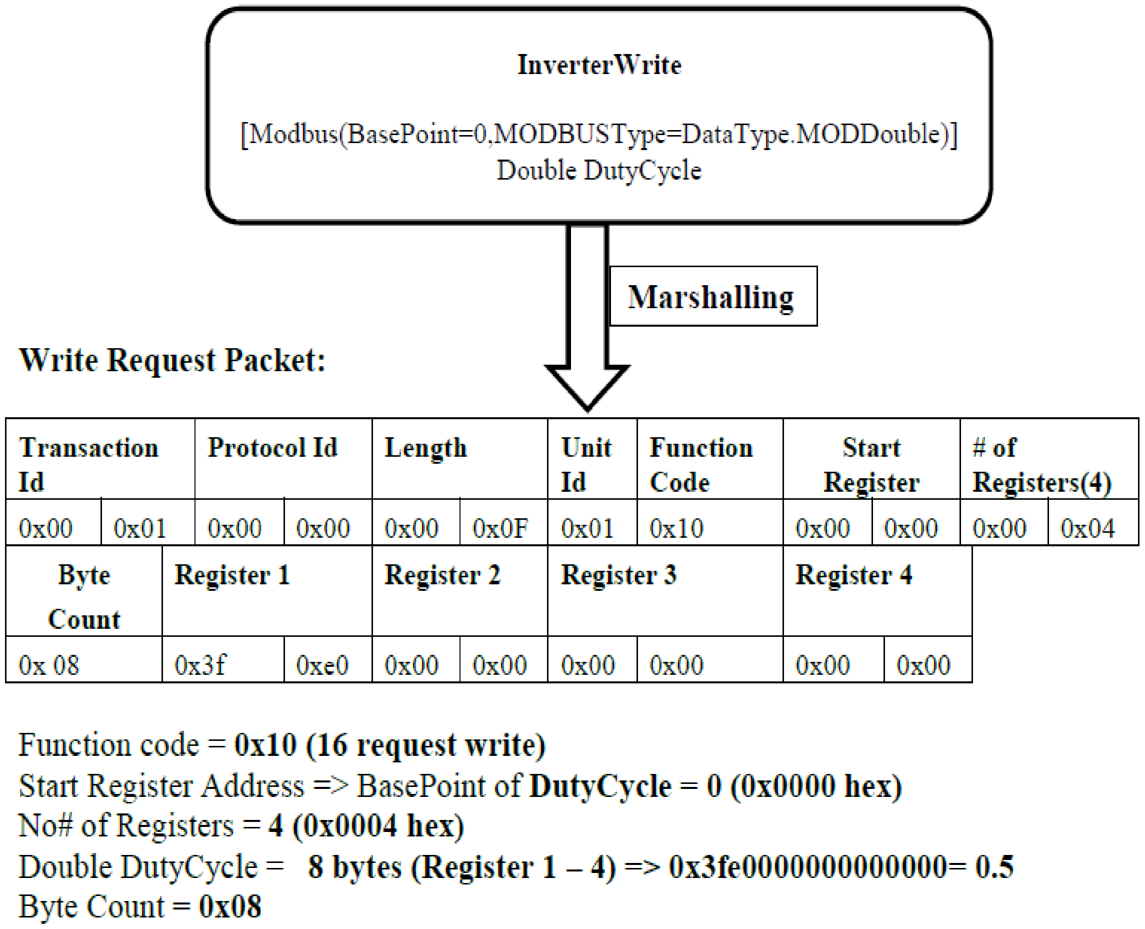
Transaction Id		Protocol Id		Length		Unit Id	Function Code	Byte count	Register 1		Register 2	
0x00	0x01	0x00	0x00	0x00	0x1B	0x01	0x03	0x18	0x40	0x72	0x89	0x25
Register 3		**Register 4**		**Register 5**		**Register 6**		**Register 7**		**Register 8**		
0xc0	0xd4	0x71	0x06	0x40	0x72	0xc0	0xd1	0xf2	0xf7	0x03	0x01	
Register 9		**Register 10**		**Register 11**		**Register 12**						
0x3f	0xf0	0x00	0x00	0x00	0x00	0x00	0x00					

Double V = **8 bytes (Register 1 – 4) => 0x40728925c0d47106 = 296.5717171**
Double I = **8 bytes (Register 5 – 8) => 0x4072c0d1f2f70301= 300.0512571**
Double simTime = **8 bytes (Register 9 – 12) => 0x3ff0000000000000= 1.0**

Figure 4.13: Unmarshalling output of the InverterRead Object

to Fig 4.4 along with the appropriate MBAP header.

Unmarshalling of InverterRead object is illustrated in Fig 4.13, after receiving appropriate well formed Modbus response read packet corresponding to Fig 4.4. The received response packet's, packet completeness and packet identification steps are processed via ModbusMarshaller's discover method (as explained in the earlier section). The ModbusMarshaller's unmarshall method is invoked which takes three parameters, firstly, an instance of transactionDescription with transactiontype as ResponseRead, secondly, InverterRead domain object and thirdly, an instance of ProtocolBuffer (that is the response packet illustrated in Fig 4.13). Based on the knowledge of the passed in parameters and its ModbusMetadata reference the InverterRead domain object data members are updated by processing the ProtocolBuffer instance.



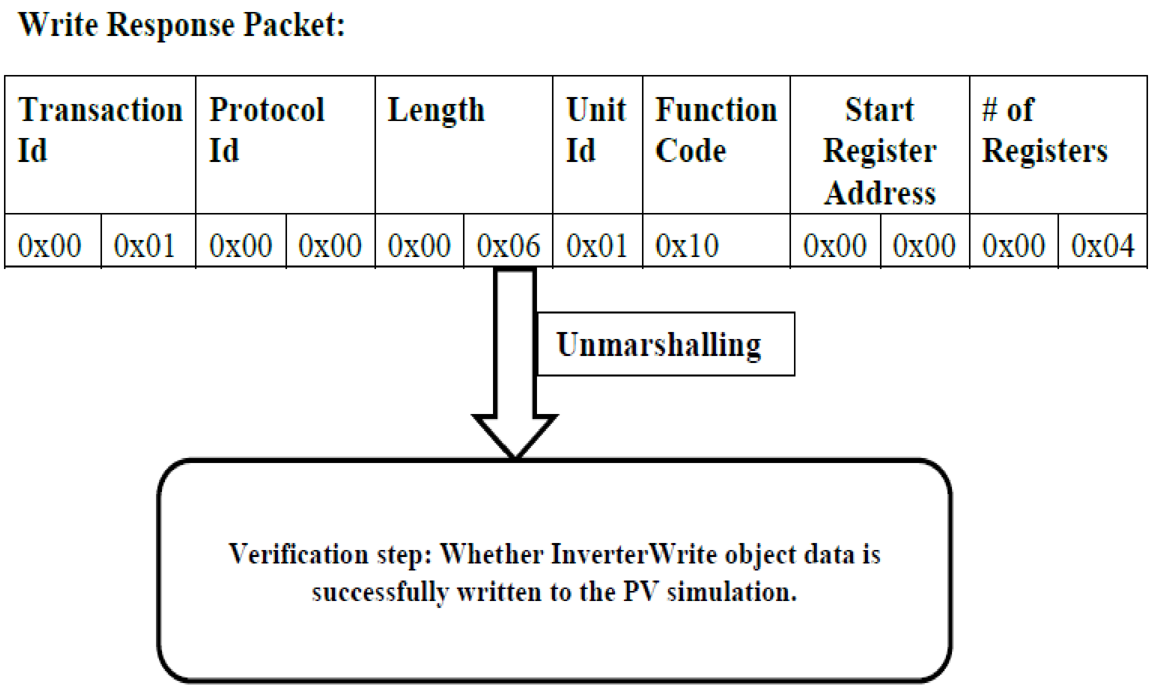
Transaction Id		Protocol Id		Length		Unit Id	Function Code	Start Register		# of Registers(4)	
0x00	0x01	0x00	0x00	0x00	0x0F	0x01	0x10	0x00	0x00	0x00	0x04

Byte Count	Register 1		Register 2		Register 3		Register 4	
0x 08	0x3f	0xe0	0x00	0x00	0x00	0x00	0x00	0x00

Function code = **0x10 (16 request write)**
Start Register Address => BasePoint of **DutyCycle = 0 (0x0000 hex)**
No# of Registers = **4 (0x0004 hex)**
Double DutyCycle = **8 bytes (Register 1 – 4) => 0x3fe0000000000000= 0.5**
Byte Count = **0x08**

Figure 4.14: Marshalling output of the InverterWrite Object

Marshalling of InverterWrite object is illustrated in Fig 4.14, which is achieved by invoking ModbusMarshaller's marshall method which takes an instance of TransactionDescriptor with transactiontype as RequestWrite (since a request to write specific values is made), InverterWrite domain object and an instance of ProtocolBuffer as parameters. Based on the knowledge of the passed in parameters and its ModbusMetadata reference the marshall method updates the instance of the ProtocolBuffer to a well formed Modbus request write packet corresponding to Fig 4.5 along with appropriate MBAP header.


Write Response Packet:
Unmarshalling
Verification step: Whether InverterWrite object data is successfully written to the PV simulation.

Transaction Id		Protocol Id		Length		Unit Id	Function Code	Start Register Address		# of Registers	
0x00	0x01	0x00	0x00	0x00	0x06	0x01	0x10	0x00	0x00	0x00	0x04

Figure 4.15: Unmarshalling output of the InverterWrite Object

Unmarshalling of InverterWrite Object is illustrated in Fig 4.15, after receiving appropriate well formed Modbus response write packet corresponding to Fig 4.5. The received response packet completeness and packet identification steps are processed via ModbusMarshaller's discover method (as explained in the earlier section). ModbusMarshaller's unmarshall method is invoked, which takes three parameters, firstly, an instance of transactionDescription with transactiontype as ResponseWrite, secondly, InverterWrite domain object and thirdly, an instance of ProtocolBuffer (that is the response packet illustrated in Fig 4.15). Based on the knowledge of the passed in parameters and its ModbusMetadata reference it is verified whether InverterWrite object values are successfully written to the simulation server or not.

4.1.4.4 Sample code work-flow of the case study from domain expert point of view

is illustrated in listing:4.3 where the domain expert instantiates the ModbusMarshaller and TransactionManager objects. The domain objects (InverterRead, InverterWrite along with their ModbusAttribute's information) are declared and initialized. The software client application is connected to the PV simulation server via TCP/IP. The TrasanctionManger's 'Read' operation is invoked to read the InverterRead data members values from the PV simulation. The InverterWrite object's dutycycle value is computed by calling MPPT control algorithm PO_ MPPT with InverterRead object data members (V, I and simTime) as parameters and the computed duty cycle is written to the PV simulation via TransactionManager's 'Write' operation.

The middleware interfaces abstracts the Modbus protocol concepts, domain object's marshalling/unmarshalling process and underneath network transaction/communication concepts from the domain expert's point of view. The domain expert solely focuses on the control algorithm implementation (P_ MPPT method implementation). In the current implementation, the amount of the domain data queried (i.e. size of the domain object) from a device in single 'Read' or 'Write' transaction is limited to the size of a single protocol packet. For example, the maximum size of modbus TCP/IP packet length is 260 bytes and the size of the domain object should be less than 260 bytes. If the domain user wants to query more data than the size of a single protocol packet then he has to declare multiple domain objects and invoke 'Read' or 'Write' operation accordingly.

Code Snippet 4.3: Sample code work-flow of the case study

```
//Instantiate the domain objects
InverterRead invRead = new InverterRead();
InverterWrite invWrite = new InverterWrite();
List<Type> lstDomainObj = new List<Type>();
lstDomainObj.Add(invRead.GetType());
lstDomainObj.Add(invWrite.GetType());
//Instantiate the ModbusMarshaller with the Domain objects "Type"
//which instantiates the ModbusMetadata instance (with ModbusPointDefinitions)
ModbusMarshaller mm = new ModbusMarshaller(lstDomainObj);
//Instantiate the TransactionManager
TransactionManager tm = new TransactionManager(mm);
//Connect to the simulation server
tm.connectClient('XXX.XXX.XXX.XXX', 502);
for (int i = 0; i < 1000; i++)
{    tm.Read(invRead);
     Thread.Sleep(3);
     // MPPT Control Algorithm
     invWrite.Duty = PO_MPPT(invRead.V, invRead.I, invRead.simTime);
     tm.Write(invWrite);
     Thread.Sleep(3);
}
tm.disconnectClient();
```

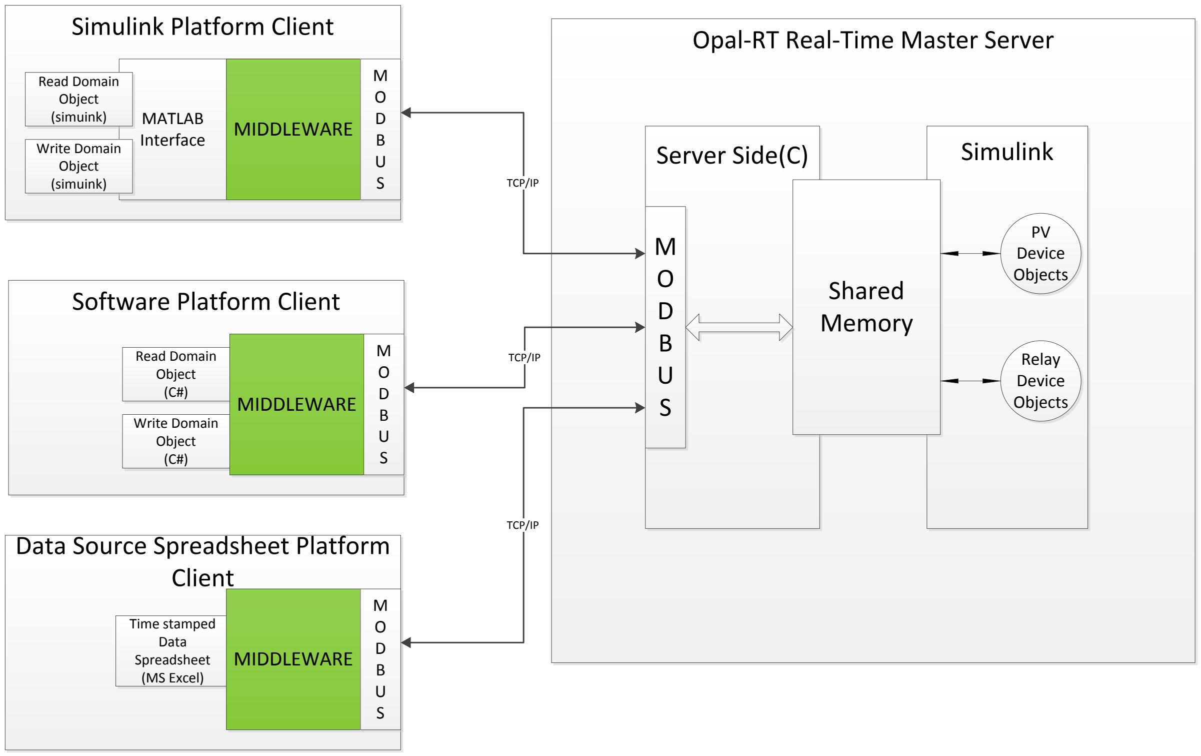


Figure 4.16: Block diagram of Multi-Client Single-Server Model [1]

4.1.5 Other distributed client communication

The 'domain-protocol mapping' approach can be extended to other platforms like simulink [36] and excel spreadsheet. The domain objects declared in respective platform representation can be marshalled to a specific protocol packet or objects can be updated (unmarshalled) from a specific protocol packet using appropriate mapping information. The middleware implemented in this study is exported as dynamic link library (dll) and used as the the interface layer in distributed multi-client communication, in [1]. Figure 4.16 illustrates the block diagram of the multi-client single server communication, where the implemented middleware is used as an interface layer for marshalling and unmarshalling the domain objects. The simulink platform client implements MPPT algorithm using state flow dynamic controllers in Matlab programming language. The domain objects are defined in simulink language, and middleware's 'Read' and 'Write' operations are used to read the data from the simulation or to write the data to the simulation. The software platform client is explained earlier. The data source client performs unidirectional communication (from client to server only) and is used to send the solar irradiance profile data values to the PV system on the server. The data values are read from a time-stamped Microsoft Excel spreadsheet, and the middleware's 'Write' operations are used to write the data to the PV system on the server.

4.2 Second Case Study: A scenario in Building energy management

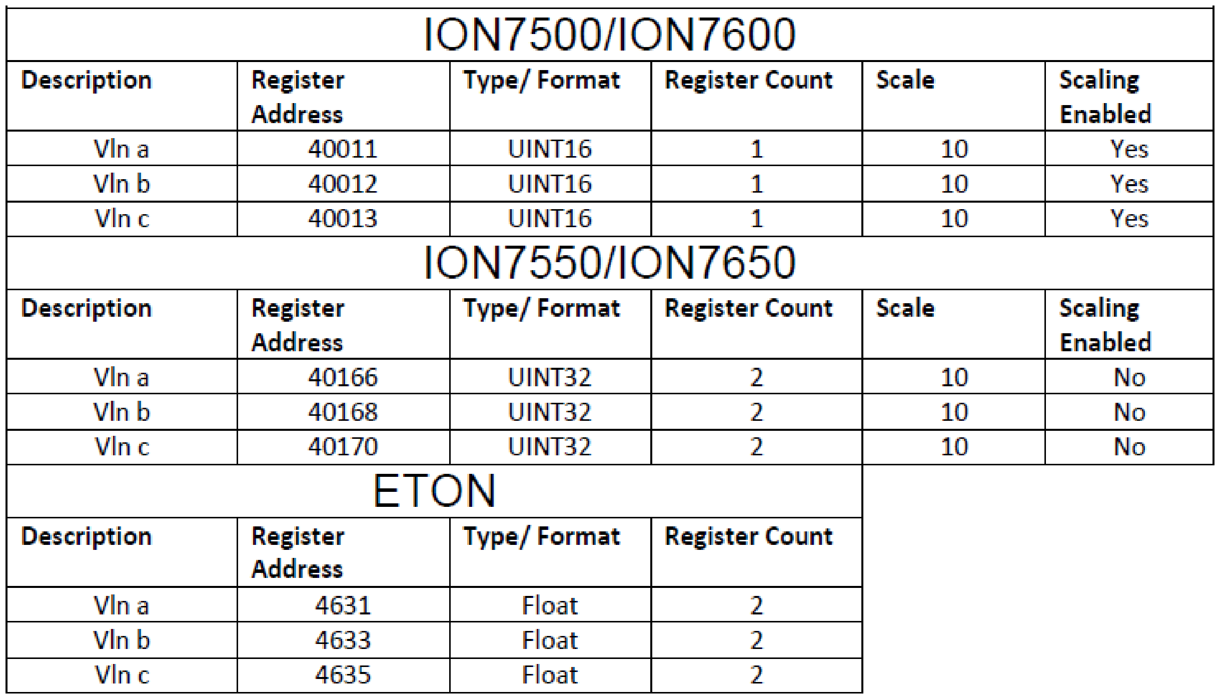
In this study, the middleware developed in the previous section is used by the power monitoring application to integrated ION [9] and ETON [10] power meters to capture the power readings using the Modbus protocol. This section begins with a brief introduction of the domain, the power meters and the description of the scenario. The code snippets of the domain objects, device metadata, domain mapping and type conversion between the domain object's data and power meter's data is also discussed.

4.2.1 Domain Background

The Building Energy Monitoring project was set up to monitor the power usage of a number of buildings on campus, correlating the power usage with occupant activity. Occupant activity was measured using wireless sensors placed on the doors of the building, counting the number of people entering or leaving the premises. The power usage was monitored using power meters mounted at the incoming power service points. These meters were a combination of ION7500/7550 and power xpert (Eaton) power meters. This study is focused on power monitoring aspect of the building energy project.

4.2.2 ION and ETON Power Meters

The ION and ETON power meters are COTS devices which provide enhanced functions for monitoring power consumption and power quality. These power meters can be used in various applications, including energy management, monitoring circuit loading, and identifying power quality problems [10]. The data representation in the ION and the ETON power



ION7500/ION7600

Description	Register Address	Type/ Format	Register Count	Scale	Scaling Enabled
Vln a	40011	UINT16	1	10	Yes
Vln b	40012	UINT16	1	10	Yes
Vln c	40013	UINT16	1	10	Yes

ION7550/ION7650

Description	Register Address	Type/ Format	Register Count	Scale	Scaling Enabled
Vln a	40166	UINT32	2	10	No
Vln b	40168	UINT32	2	10	No
Vln c	40170	UINT32	2	10	No

ETON

Description	Register Address	Type/ Format	Register Count
Vln a	4631	Float	2
Vln b	4633	Float	2
Vln c	4635	Float	2

Figure 4.17: Phase voltage representation in Modbus enabled ION and ETON meters

devices is vendor specific. The support to access the device data is provided via industry standard protocols MODBUS and SNMP. Figure 4.17 illustrates device metadata for phase voltage data, where the ION and ETON power meters are configured as Modbus slave devices. The Modbus register map and the data representation on the ION and ETON meters is different as illustrated in Fig 4.17.

4.2.3 Case study scenario description

Figure 4.18 illustrates a scenario, where the power monitoring application queries the phase voltage data readings from Modbus enabled ION7500, ION7550 and Power Xpert (Eton) power meters. The power monitoring application expects the voltage data in double data type. However, the 3-phase voltage data is represented as UINT16, UINT32 and floating point in ION7500, ION7550 and Power Xpert (Eton) respectively. The Modbus register map of the voltage data varies among the meters. Therefore, the middleware that is used by the application to query the data needs to address the device heterogeneity caused by

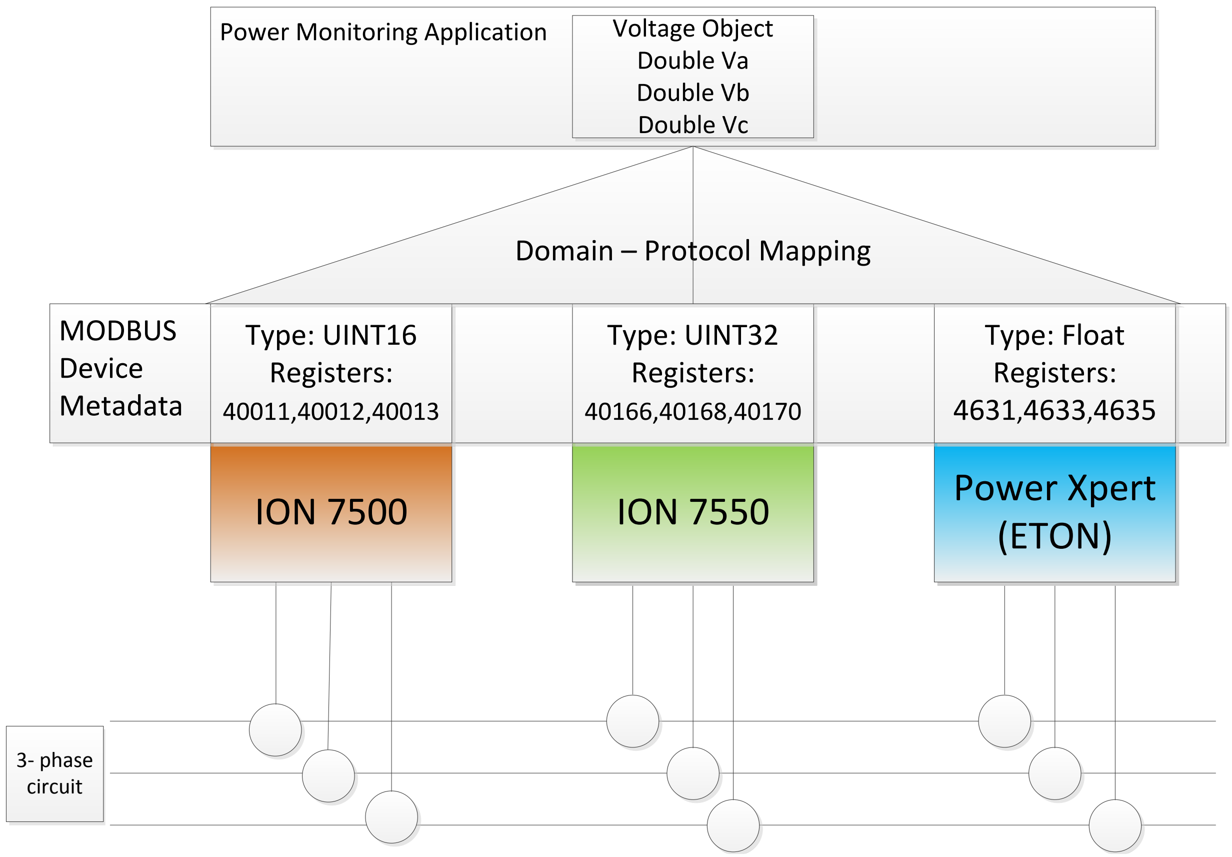


Figure 4.18: Power Monitoring scenario description

integrating different types of power meters. It should also the address the type conversion concern between domain object's data and the data provided by the power meter. The proposed middleware address the device heterogeneity by providing a high level mapping between the domain object and the device metadata. The middleware provides appropriate interfaces to address the type conversion concern, and this process is abstracted from the domain expert, since the conversion occurs during 'Read' or 'Write' operation.

Code Snippet 4.4: Voltage domain object

```
class Voltage {
    Double vlna;
        Double vlnb;
        Double vlnc
        public Double Vlna {
            get { return vlna; }
            set { vlna = value; }
        }
        public Double Vlnb {
            get { return vlnb; }
            set { vlnb = value; }
        }
        public Double Vlnc {
            get { return vlnc; }
            set { vlnc = value; }
        }
    }
```

Code Snippet 4.5: Domain object - Protocol mapping for ION7500 power meter

```
class Voltage {
    Double vlna;
        Double vlnb;
        Double vlnc
        [Modbus(BasePoint = 10,
         MODBUSType = ModbusPointDefinition.DataType.UINT16, Scale = 10)]
        public Double Vlna {
            get { return vlna; }
            set { vlna = value; }
        }

        [Modbus(BasePoint = 11,
         MODBUSType = ModbusPointDefinition.DataType.UINT16, Scale = 10)]
        public Double Vlnb {
            get { return vlnb; }
            set { vlnb = value; }
        }

        [Modbus(BasePoint = 12,
         MODBUSType = ModbusPointDefinition.DataType.UINT16, Scale = 10)]
        public Double Vlnc {
            get { return vlnc; }
            set { vlnc = value; }
        }
    }
```

4.2.4 Domain Object, protocol mapping and type conversion illustration

Listing:4.4 illustrates the voltage domain object which encapsulate the 3-phase voltage data in double data type. The domain object - protocol mapping for ION7500 is illustrated in Listing: 4.5. The mapping for ION7550 and Power Xpert meter is illustrated in the appendix section. It is evident from the Listing: 4.5 that the domain object notion of the voltage is represented in double and the data provided by the power meter is in UINT16. The ModbusTypeConverter class illustrated in Fig 4.9 provide operations such as 'fromType' and 'toType', These operations use .NET's *Convert* base class operations for appropriate type conversion between domain object and the data provided by the device at run-time using reflection. The TransactionManager's 'Read' operation is invoked by the power monitoring application to read the voltage data from the power meters.

Chapter 5

Conclusion and Future Work

In this work, a 'domain-protocol mapping' technique is proposed and implemented as a middleware framework. A high level design model of the middleware is proposed, which provides the abstract entities and their corresponding interfaces, which should be specialized to provide core functionality for a family of devices and their respective communication protocols. The detail level design class model of the middleware is implemented using .NET 4.5 framework and C# programming language for Modbus protocol supported devices.

The middleware provides 'Read' and 'Write' operations that are used by the domain expert to read or write the domain data to the device using specialized protocol adapter. The 'Read' and 'Write' operations abstracts complete marshalling and unmarshalling process of the domain object from a domain expert who may or may not be a software engineer. The developed middleware is demonstrated by two case-studies which highlights the following benefits of the mapping approach:

- Modbus protocol adapter is reused across distributed network control power system and power monitoring applications.
- The domain experts makes the modifications to the domain objects (to include a new data point) in both the case studies, and no modifications are done to the Modbus protocol adapter.
- The device and software heterogeneity caused by integrating different COTS power meters is isolated in the mapping layer by providing a high level mapping between the domain object and the devices.
- The middleware is also used as an interface layer for reading or writing the domain objects to the simulation server in other client platforms such as simulink and excel

spreadsheet.

- The domain expert (control engineer) focuses on MPPT algorithms implementation rather addressing communication protocol implementation details.

5.1 Future work

Currently, the device metadata specification is performed in an implicit manner by decorating the domain object's data members by C# decorated attributes. In future work more explicit way of specifying the device metadata can be explored by using configuration files.

In the current implementation, the amount of the domain data queried (i.e. size of the domain object) from a device in single 'Read' or 'Write' transaction is limited to the size of a single protocol packet which can be overcome by encapsulating data in multiple domain objects. However, more detailed approach for subtransaction management during marshalling and unmarshalling process can be examined in future work.

Currently, the middleware supports Modbus protocol enabled devices and TCP/IP network end point. In future work, the middleware framework can be extended to support multiple protocols and other network end points such as serial and wireless network interfaces can be examined.

The case-study applications and the middleware implemented in this work is based on single threaded model i.e. a 'Read' or 'Write' operation waits for a response from the device before proceeding further. However, in more complex multi-threaded model based applications, the middleware should employ the multi-threaded model, where the independent threads are responsible for notifying the completion of the 'Read' or 'Write' operations to the user/application.

Finally, the ’domain-protocol’ mapping approach can be explored within an existing middleware and model driven engineering solutions to address device and software heterogeneity challenges in distributed application development involving multi-disciplinary teams.

References

[1] A. Jain, “Synchronized Real-time Simulation of Distributed Networked Controls for a Power System Case Study,” Master’s thesis, Colorado State University, 2013.

[2] I. Modbus, “Modbus Application Protocol Specification v1. 1b3,” *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2012.

[3] L. Coetzee and J. Eksteen, “The Internet of Things - promise for the future? An introduction,” in *IST-Africa Conference Proceedings, 2011*, pp. 1–9, 2011.

[4] R. Gupta and M.-Y. Chow, “Networked Control System: Overview and Research Trends,” *Industrial Electronics, IEEE Transactions on*, vol. 57, no. 7, pp. 2527–2535, 2010.

[5] V. Potdar, A. Sharif, and E. Chang, “Wireless Sensor Networks: A Survey,” in *Advanced Information Networking and Applications Workshops, 2009. WAINA’09. International Conference on*, pp. 636–641, IEEE, May 2009.

[6] I. Modbus, “Modbus Messaging on TCP/IP Implementation Guide v1. 0b,” *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2006.

[7] K. Curtis, “A DNP3 Protocol Primer,” *DNP User Group*, 2005.

[8] J. Case, M. Fedor, M. Schoffstall, and C. Davin, *A Simple Network Management Protocol (SNMP).* Network Information Center, SRI International, 1989.

[9] PowerLogic, “Modbus and ION Technology,” 2014. [Online; Accessed Jan 14, 2014].

[10] EATON, “EATON Power meters,” 2014. [Online; Accessed Jan 14, 2014].

[11] J. Linwood and D. Minter, “An Overview of Mapping,” in *Beginning Hibernate*, pp. 77–90, Springer, 2010.

[12] “Types of Middleware.” Available at url http://apprenda.com/library/architecture/types-of-middleware/ (2014/01/09).

[13] S. Vinoski, “Where is Middleware?,” *IEEE Internet Computing*, vol. 6, no. 2, pp. 83–85, 2002.

[14] M. Molla and S. Ahamed, “A Survey of Middleware for Sensor Network and Challenges,” in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pp. 6 pp.–228, 2006.

[15] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless Sensor Networks: A Survey,” *Computer Networks*, vol. 38, pp. 393–422, 2002.

[16] B. Elen, S. Michiels, W. Joosen, and P. Verbaeten, “A Middleware Pattern to Support Complex Sensor Network Applications,” *status: published*, 2006.

[17] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo, "Middleware to Support Sensor Network Applications," *Network, IEEE*, vol. 18, no. 1, pp. 6–14, 2004.

[18] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, "A Message-Oriented Middleware for Sensor Networks," in *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pp. 127–134, ACM, 2004.

[19] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[20] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The Stanford Data Stream Management System," *Book chapter*, 2004.

[21] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006.

[22] L. Gurgen, C. Roncancio, C. Labbé, A. Bottaro, and V. Olive, "SStreaMWare: a Service Oriented Middleware for Heterogeneous Sensor Data Management," in *Proceedings of the 5th international conference on Pervasive services*, pp. 121–130, ACM, 2008.

[23] M. Brito, L. Vale, P. Carvalho, and J. Henriques, "A Sensor Middleware for Integration of Heterogeneous Medical Devices," in *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pp. 5189–5192, IEEE, 2010.

[24] K. Aberer, M. Hauswirth, and A. Salehi, "Global Sensor Networks," *EPFL, Lausanne, Tech. Rep*, 2006.

[25] K.-D. Kim and P. Kumar, "The Importance, Design and Implementation of a Middleware for Networked Control Systems," in *Networked Control Systems*, pp. 1–29, Springer, 2010.

[26] G. Baliga, S. Graham, and P. Kumar, "Middleware and Abstractions in the Convergence of Control with Communication and Computation," in *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pp. 4245–4250, IEEE, 2005.

[27] A. Silberschatz, P. B. Galvin, and G. Gagne, *Applied Operating System Concepts*. Wiley, 2003.

[28] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta, L. Mostarda, and H. Muccini, "A model-driven engineering framework for architecting and analysing Wireless Sensor Networks," in *Software Engineering for Sensor Network Applications (SESENA), 2012 Third International Workshop on*, pp. 1–7, IEEE, 2012.

[29] A. Di Marco and S. Pace, “Model-driven approach to Agilla Agent generation,” in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pp. 1482–1487, IEEE, 2013.

[30] C.-L. Fok, G.-C. Roman, and C. Lu, “Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 3, p. 16, 2009.

[31] F. Fleurey, B. Morin, A. Solberg, and O. Barais, “MDE to Manage Communications with and between Resource-Constrained Systems,” in *Model Driven Engineering Languages and Systems*, pp. 349–363, Springer, 2011.

[32] Opal-RT Technologies Inc., *RT-LAB User Guide*, ver.10.4 ed., 2007.

[33] D. Hohm and M. Ropp, “Comparative Study of Maximum Power Point Tracking Algorithms Using an Experimental, Programmable, Maximum Power Point Tracking Test Bed,” in *Photovoltaic Specialists Conference, 2000. Conference Record of the Twenty-Eighth IEEE*, pp. 1699–1702, IEEE, 2000.

[34] J. J. Nedumgatt, K. Jayakrishnan, S. Umashankar, D. Vijayakumar, and D. Kothari, “Perturb and Observe MPPT Algorithm for Solar PV Systems-Modeling and Simulation,” in *India Conference (INDICON), 2011 Annual IEEE*, pp. 1–6, IEEE, 2011.

[35] A. Mackey, W. Tulloch, and M. Krishnan, *Introducing .NET 4.5*. Apress, 2012.

[36] J. B. Dabney and T. L. Harman, *Mastering Simulink 4*. Prentice Hall PTR, 2001.

Appendix A

Sample device metadata specification code snippets for power meters

A.1 Sample Protocol mapping code snippets for power meters

The domain object - protocol mapping for ION7550 and Power Xpert power meters is illustrated in Listings: A.1 and A.2.

Code Snippet A.1: Domain object - Protocol mapping for ION7550 power meter

```
class Voltage{
    Double vlna;
        Double vlnb;
        Double vlnc
        [Modbus(BasePoint = 165,
         MODBUSType = ModbusPointDefinition.DataType.UINT32)]
        public Double Vlna {
            get { return vlna; }
            set { vlna = value; }
        }
        [Modbus(BasePoint = 167,
         MODBUSType = ModbusPointDefinition.DataType.UINT32)]
        public Double Vlnb {
            get { return vlnb; }
            set { vlnb = value; }
        }
        [Modbus(BasePoint = 169,
         MODBUSType = ModbusPointDefinition.DataType.UINT32)]
        public Double Vlnc {
            get { return vlnc; }
            set { vlnc = value; }
        }
    }
```

Code Snippet A.2: Domain object - Protocol mapping for ETON power meter

```
class Voltage{
    Double vlna;
        Double vlnb;
        Double vlnc
        [Modbus(BasePoint = 4631,
         MODBUSType = ModbusPointDefinition.DataType.Float)]
        public Double Vlna {
            get { return vlna; }
            set { vlna = value; }
        }
        [Modbus(BasePoint = 4633,
         MODBUSType = ModbusPointDefinition.DataType.Float)]
        public Double Vlnb {
            get { return vlnb; }
            set { vlnb = value; }
        }
        [Modbus(BasePoint = 4635,
         MODBUSType = ModbusPointDefinition.DataType.Float)]
        public Double Vlnc {
            get { return vlnc; }
            set { vlnc = value; }
        }
    }
```