A Generalized Binding Framework for the Low Level Reader Protocol (LLRP)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 1 3 2008
LIBRARIES

by

Dimitrios Poulopoulos

S.B., E.E.C.S. MIT, 2007

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2008

©Dimitrios Poulopoulos, MMVIII. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part

Author ....
Department of Electrical Engineering and Computer Science
June 2008

Certified by. ....
Abel Sanchez
Research Scientist, Laboratory of Manufacturing and Productivity
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 1 3 2008
LIBRARIES

BARKER

A Generalized Binding Framework for the Low Level Reader Protocol (LLRP)

by

Dimitrios Poulopoulos

S.B., E.E.C.S. MIT, 2007

Submitted to the Department of Electrical Engineering and Computer Science
on June 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This Master of Engineering Thesis describes the design, implementation and testing of an XML binding framework for the RFID Low Level Reader Protocol (LLRP). LLRP is a recently released protocol which standardizes the interface between RFID readers and RFID middleware. The proposed framework serializes wire objects to the schema of the LLRP binary messages and parameters. The framework also validates the produced XML elements against the LLRP data model. The framework includes a data serialization mechanism on the reader's side and allows for easy and efficient data updates as an RFID Network simulator.

Thesis Supervisor: Abel Sanchez
Title: Research Scientist, Laboratory of Manufacturing and Productivity

Acknowledgments

This essay was improved by conversations with a large number of people who helped me debug it. I would like to thank Dr. Abel Sanchez for giving me the opportunity to work with the Auto-ID Labs over the last year. I have learned a lot during this time, and I was very fortunate to have his precious guidance. I would also like to thank my labmate Ane Fabo for her valuable feedback and Fivos Constantinou who helped me with some of the design decisions of my project. Next, I would like to thank Caroline Huber for her consistent support and encouragement. Finally, I would like to thank my parents for financial and psychological support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Radio Frequency Identification

1.1.1 General

Radio Frequency Identification (*RFID*) is an emerging technology which continues to be more widely adopted in numerous applications, such as access control, product identification and tracking. RFID technology has important applications in the supply chain of products in industry. RFID tagged products are identified by unique Electronic Product Codes (EPCs) as they move through the supply chain. To further enhance the manufacturing process, a mechanism is needed to record business events and data associated with a product. This mechanism needs to be able to maintain static data for a product, such as product type, date of manufacture, expiration date, etc. It should also keep track of transactional data, which include records of a product's location at a given time, or information relating a product with a specific business transaction. Finally, this mechanism should allow product discovery and offer a way to query data that is generated by others. This mechanism is currently being designed and standardized, with the goal of creating the *EPCGlobal* network. [1]

1.1.2 RFID Tags

To identify a product, the EPC is stored in a microchip attached to an antenna and attached to the product. The device, which consists of the microchip and the antenna, is known as RFID tag. The tag is capable of transmitting an RF signal containing the EPC number to a device called reader or interrogator. The reader extracts the information from the modulated signal and delivers the data to the system or application that makes use of it.

An RFID system consists of a number of readers covering a certain area of space in which the presence of tagged items needs to be controlled. The information of the tags is delivered to some host system, which keeps track of all items that are in the area of interest. This information may be stored or processed, and it may help to make business decisions or automate processes, depending on the application.

RFID Tags come in various form factors and can be easily attached to a variety of materials. They are classified according to the manner in which they get the power necessary to transmit the information. [6]

Passive Tags

Passive tags draw power from the reader. The reader generates an electromagnetic field around its antenna, which is captured by the tag. This electromagnetic field induces a voltage between the two terminals of the tag's antenna. This signal is the only power source of the tag. It must be enough to switch on the IC that retrieves the identification information from memory and to transmit a response back to the reader. This response is an amplitude modulated electromagnetic wave, according to the digital information stored in the tag memory. There are two basic mechanisms of generating this AM response signal: inductive coupling and backscatter.

Backscatter tags basically reflect part of the signal received back to the reader. Some of the power received is used to run the circuitry that retrieves the data stored in memory and controls the modulation according to that data. The rest of the energy is just reflected back to the reader. Amplitude modulation of the response wave is performed by controlling the amount of energy that is reflected.

Passive tags do not need any maintenance, and they last for years once attached to a product. Due to their low cost and longer read-range than the ones that work under inductive coupling principle, passive backscatter tags are the ones considered for supply chain applications or in general, low-price items labeling. Actually, the EPCGlobal protocol that standardizes the interaction between tags and readers, refers exclusively to this type of tags (UHF Class 1 Generation 2).

In the following chapters, when talking of RFID tags, we will be referring to passive backscatter tags (unless otherwise specified).

Semi-passive Tags

Semi-passive tags backscatter the wave coming from the reader, just as passive tags do. The difference is that they have an onboard power source to run the circuitry. This allows a higher proportion of the incoming signal to be reflected, and so, a longer read range is achieved.

Active Tags

Active tags are able to transmit the information by themselves without requiring energy from the readers. Active tags are sophisticated wireless devices, much bigger in size and more expensive. Usually, they have further functionality than just transmitting the identification number of the item to which they are attached. The memory has higher capacity and stores data of sensors built onboard. Active tags can transmit this information as far as several dozens of meters.

Active RFID solutions are being used to track valuable assets in the healthcare, manufacturing and logistics markets. But their cost, their size and their need for maintenance, make this solutions not suitable for supply chain applications.[7]

1.1.3 RFID Readers

An RFID reader is a device capable of getting the EPC data stored in a tag's memory when the tag is located close enough to its antenna. As already explained, for passive backscatter tags, the reader broadcasts an RF signal and captures the AM wave reflected back by each of the tags. By properly demodulating this response wave, the reader must be able to extract the EPC data transmitted by the tag.

The region of space around the reader's antenna, in which the broadcasted signal is powerful enough for a tag to generate a response that can be detected back by the reader, is known as the field of view (FOV) of the reader.

The read range is typically is about 5 meters and requires no line of sight between the tag and the reader. Electromagnetic radiation can go through most obstacles with little attenuation, except through conductor materials. Metallic objects totally reflect waves, affecting the FOV and causing interferences when the reflections reach back to the reader's antenna. The only possible solution is to carefully choose the location of readers and tags. Metallic objects should be avoided between them, but notice that the reflections can also increase the amount of power received by the tag when the layout is properly designed.

Metallic objects are not the only cause of interference in RFID systems. When various readers cover the same area, tags are not capable of resolving which requests they should be answering to. The layout of the system also plays an important role in avoiding this type of interference. However, there are more sophisticated mechanisms to keep this problem under control. These mechanisms basically consist of algorithms that dynamically adapt the power transmitted by the different devices, as well as scheduling the transmissions to occur separately in time.

When different tags transmit simultaneously, the reader may not be able to decode the information coming from each of the tags. If the reader has the ability to manage tags' transmissions, this type of interference can be avoided. And so, readers and tags must exchange control commands, as well as the identification information itself or the signals that carry power for tags' operation. This problem reveals the necessity of a radio protocol of certain complexity.

RFID readers achieve high read rates. A reader located at the entrance of a warehouse is capable of reading all the tags inside a truck in the time it takes the vehicle to cross the entrance. This is clearly one of the big advantages of RFID technology, versus the optical systems used today. Other advantages are the longer read range and that no line of sight is required. In addition, the memory capacity of the tags allows storing much more information than a regular bar code can contain. RFID technology allows item level identification, meaning that a serial number identifies the specific item, not just the type or family of products to which it belongs. However, the cost and complexity of RFID systems is not negligible nowadays. Will RFID eventually substitute the bar code system? It is hard to argue about this view, however both bar codes and RFID tags will coexist for a long period of time. [6]

1.1.4 Applications of RFID Technology

RFID is not a new technology. It has been used in many applications for more than ten years now. Applications of automated identification are manifold.

Automatic payment in motorways is a popular example. Users are identified and charged directly to their accounts when crossing the toll gate. The United States Department of Defense has been using RFID for tracking its containers since 1994 and losses have been cut by 90%. Airlines are starting to substitute barcodes with RFID tags for baggage handling. RFID systems are more robust as tags can be correctly detected even if they get wet or dirty or even if they are not directly facing the reader's antenna. This allows more reliable tracking of every piece of luggage, reducing losses. Higher read rates speed up the process of loading the bags into the hold of the plane or looking for the ones that need to be taken out. RFID based anti-theft systems were introduced about a decade ago and today are used almost everywhere. A plastic anti-theft tag is attached to the item, and if it is not removed or disabled by the cashier at the moment of payment, it fires an alarm when detected by the readers at the exit of the store.

The point of mentioning the previous examples was just to make the reader think of the very wide variety of businesses in which RFID can be useful. Due to its characteristics, RFID increases reliability and speeds up processes in many Auto-ID applications in which barcodes are still being used (e.g. baggage handling). Moreover, it opens a new range of possibilities, such as the automatic toll payment. [6]

1.2 Overview of the EPC Network and its Protocol Stack

EPCGlobal is the organization that leads the development of all the standards for RFID systems. All of them are open and can be downloaded for free from their website. It is a not-for-profit organization, entrusted by industry, with the aim of establishing and supporting RFID technology in the supply chain.

For supply chain applications, in which the same item may be read by many different organizations and recognized as "the same thing", it is of vital importance to agree in a universal numbering system. Such a system is not necessary in a "closed" RFID application, e.g. automatic toll payment, as anybody but the corresponding company is going to capture and use that information. But when it is about tracking items all through the supply chain, each item's ID must be recognized by all parties from the manufacturer to the retailer. The universal serial number that identifies each product is called EPC (Electronic Product Code). [6]

The protocol that defines the communication between tags and readers is UHF Class 1 Generation 2. There is another protocol (HF Gen 2), still in development, which will standardize that same interface for a different frequency range. Today there are many tag and reader manufacturers. Using a standard air protocol assures every device will be able to read every tag. [5]

The main purpose of an RFID reader is to capture the tag data in its field-of-view. Readers are not designed for storing or processing that data. They just deliver it to some higher level system (it might be a physical device or a piece of software) that has those functions.

An RFID reader may not be continuously looking for tags in its field of view. The reader may "read" only when receiving a certain command or just "read" periodically. In conclusion, that higher level system, which collects data, will also be in charge of configuring and controlling readers' operations.

The Low Level Reader Protocol (LLRP) standardizes the interface between readers and higher level RFID systems, often referred to as RFID middleware [5]. Tags, readers and the intermediate software form the ID system are depicted in Figure 1-1.

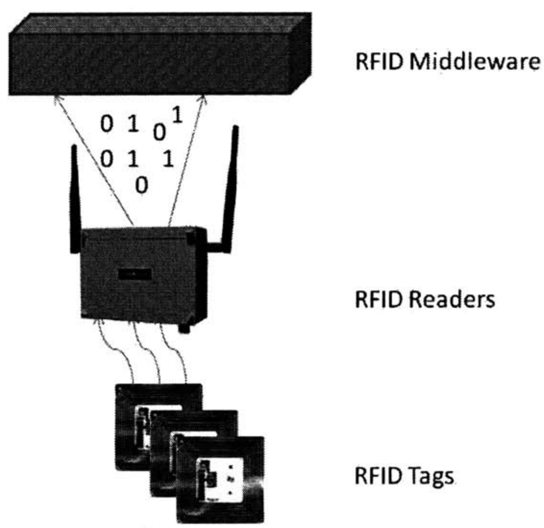

Figure 1-1: RFID system. Readers and tags exchange information embedded in RF signals. Readers extract the digital information and deliver it to the host system in a binary format.

As already mentioned in the introduction, for supply chain management purposes, systems of manufacturers, distributors and retailers must be interconnected. That way, maximum benefit can be derived from all data generated. But as we pointed out earlier, this requires a standard way of storing and exchanging information. EPCIS

(EPC Information Services) is the main EPCGlobal standard for sharing information generated by RFID systems.

The interconnection of all these systems forms a great network, the EPC Network, sometimes also called the "Internet of Things." A simplified diagram of the network is shown above. Through the EPC network, manufacturers, distributors and retailers can access the data generated by the ID systems of any other company with which they do business. All that information that is exchanged is encoded into XML files, following the schemas defined by the EPCIS standard.

Somewhere in that network there is a record for each time an item has been detected in a certain location. Let's say we have access to the network and we are looking for a specific item. We do have its EPC number, but do not know to which final ID system we need to address our request. Just as the Domain Name System (DNS) does when we are looking for a webpage without its IP, we need a higher level entity that handles our request and redirects it to its destination. Let's consider now that we are interested in every location a certain item has been detected. The response to our request comprises information not from one, but from several databases in the network. The aim of the discovery layer is to handle these types of queries, which require a higher level centralized system connected to all other nodes in the network.[3]

In the following sections of this chapter I will explain in greater detail some of the interfaces that I just introduced. From the lowest level (the numbering system and the air protocol) to the highest level standard relevant to this project (EPCIS).

1.3 Organization of this Thesis

Chapter 2 gives some background on LLRP, specifically the previous work in message and parameter encoding. Also, Chapter 2 describes the high level characteristics of the proposed framework. The XML serialization and validation processes are described in Chapter 3. Chapter 3 ends with explaining how the testing was performed. Chapter 4 describes the advantages of the XML binding and some future work that would improve the project. Finally, Chapter 5 contains a discussion of the conclusions we reached implementing the Binding Framework of the LLRP Library.

Chapter 2

Background

2.1 Previous Work on LLRP

LLRP is an application layer, message-oriented protocol, which standardizes the formats and procedures of communication between Clients and Readers. Using the LLRP interface the Client can retrieve and change the Reader configuration, controlling in this way the Readers operation. The functionality provided by the interface is summarized below:

- Allows Clients to retrieve Reader device Capabilities
- Allows Clients to control Readers to inventory, read or write tags and execute other access commands
- Allows Clients to control the Reader device operation (e.g. Power levels, spectrum utilization, etc.)
- Provides robust status reporting and error handling
- Allows future expansions for support of additional air protocols
- Allows Reader vendors to define custom vendor-specific extensions

Using LLRP, clients and readers exchange protocol data units (PDUs) called messages. LLRP specification is a collection of messages for different purposes. The communication between clients and readers is fundamentally of a request-response type. The specification defines what the requests are and the expected response from the reader for each of them. The messages contain fields and parameters with further details of the requested or performed LLRP operation. There are a few messages that the reader can send by itself, basically reports and certain events notifications.[5]

Messages are encoded into a binary stream and sent over a TCP connection. Figure 2-1 shows the structure of a frame exchanged between a client and reader implementing the LLRP interface. Link (802.11, Ethernet), network (IP) and transport (TCP) layer information is added to the LLRP Message so that it can be sent over the network.

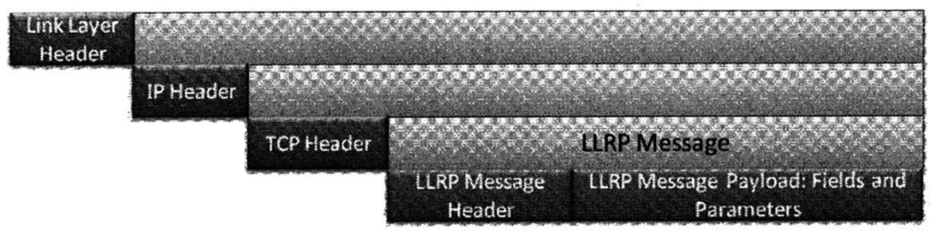

Figure 2-1: Structure of a frame exchanged between and a client and a reader implementing the LLRP interface.

Any client or reader can be addressed given its network end point: IP address and port number. Although it is not compulsory, the specification sets 5084 as the default port number both for the client and the reader. Notice that the exchange of LLRP messages occurs over a TCP channel, which is a connection-oriented transport protocol. As a consequence, the first step in any client-reader communication is always connection establishment.

2.2 LLRP Messages

LLRP messages are the protocol data units by which a Client controls Reader operation. A Client can send messages to perform one of the following:

- Query Reader Capabilities

- Control the Reader's air protocol inventory and RF operations
- Control the tag access operations performed by the Reader
- Query/Set Reader configuration, and close the LLRP connection

Messages sent by a Client can change a Readers state and, since state consistency is essential for the system to function properly, all Client messages must be acknowledged from the Reader. Consequently, Reader messages are primarily responses to Client messages. Additionally, Readers can send the following messages:

- Reports from Reader to Client. Reports include Reader device status, tag data, RF analysis report
- Errors. Reader responds with a generic error message when it receives an unsupported message type

An LLRP compliant Client must be capable of sending all Client messages defined in the LLRP specification and receive all Reader messages. Similarly, LLRP compliant Readers must handle Client messages properly and be able to send all Reader messages. As defined in the LLRP specification a Message contains the following:

- Version Value
- Message Type
- Message ID
- A variable number of optional or mandatory Parameters

This implementation defines all LLRP messages using an object model and can, therefore, be used to implement both Client and Reader logic. Each message is implemented as a concrete class, which implements the ILlrpMessage interface. The structure of the Message classes, which is common for all Messages, is described below:

Each LLRP Message class contains the appropriate Parameters, as defined in the LLRP specification. Parameters are objects which implement the ILlrpParameter interface. A valid Message can be constructed by passing in all the mandatory and/or optional Parameters to the Message constructor. To ensure data encapsulation, Parameters are defined as private member variables and are made accessible as public properties. Properties are a built-in mechanism in C#, which allows access to the data fields through get/set methods. Message Parameters can be accessed and modified at any time after a Message is constructed.[9]

In addition to Parameters, Message classes contain a private member variable of type MessageEncoding. The MessageEncoding class represents the binary encoding of the Message and provides methods for accessing and setting the Message header information, i.e. version value, Message type and Message ID.

2.3 LLRP Parameters

LLRP Parameters are the mechanism by which Messages communicate the details of LLRP operation. As defined in the LLRP specification each Parameter contains the following:

- Parameter Type
- Individual Fields or Sub-parameters

The structure of LLRP Parameters in this implementation is similar to LLRP Messages. Each LLRP Parameter is implemented as a concrete class which implements the ILlrpParameter interface. Similar to Messages, Parameters contain the appropriate fields or sub-Parameters and provide the same construction and access methods.

There are two different encodings for LLRP Parameters, thus two classes to represent the binary encoding: TLVEncoding and TVEncoding. All Parameters include a field of one of the two encoding types. Both Parameter encoding classes implement the ParameterEncoding interface. All LLRP Parameters implement the ILlrpParameter interface, which defines a single method called GetParameterEncoding(). The GetParameterEncoding() method returns an object of type ParameterEncoding.[9]

2.4 Current LLRP Client

The Client implementation provides functionality for both initiating and accepting a connection. The Client can start listening for a connection on a port that is specified by the user. The GUI also allows a user to specify the IP address and port number of a Reader and attempt to initiate a connection. In order to establish a connection the specified Reader must be listening for incoming connections.

After a TCP connection is established the Reader must reply with a status report Message. The report must include a ConnectionAttemptEvent Parameter and should indicate connection success if the TCP connection was established successfully.

The Client implementation uses asynchronous operations to communicate with Readers. When the Client is set to listen for incoming connections it starts an asynchronous listen operation, using the build-in asynchronous operations of the TcpListener class, which are provided by the .NET framework. The asynchronous listen operation specifies a Callback method, which is called upon a connection attempt.

The OnConnect() callback method is called once a TCP connection between the Client and a Reader is established. The Client then starts an asynchronous Read operation, waiting for a ReaderEventNotification Message from the Reader. If this operation times out, an incorrect Message is received, or the status report indicates an unsuccessful connection, then the TCP Connection is terminated.

An LLRP Client would normally connect to and control the operation of multiple Readers. This implementation, however, currently only supports a single Reader. The GUI allows the user to initiate a connection to only a single Reader. Additionally, if the Client is in listening mode, once the Client accepts a connection it stops listening and ignores any other connection attempts.[9]

2.5 Characteristics of the Binding Framework

Our framework serializes top-level objects according to the specifications of the LLRP binary messages and parameters. It also validates the produced XML elements against the LLRP data model. The use of this framework provides a data serialization mechanism at the reader's side and allows for easy and efficient data update when simulating the RFID Network. As RFID technology continues to be more widely adopted, it becomes imperative to standardize the way information is stored at the reader's side. A standard interface will allow RFID systems to be designed without dependence on vendor proprietary interfaces and will simplify their implementation. It is envisioned that LLRP will soon be adopted as the standard Reader - Client interface.

The implementation of the proposed framework was oriented towards reducing the complexity of LLRP, which primarily lies in the binary nested format used for representing LLRP Messages and Parameters. Our system uses the nested object model that is used to represent all the Messages an Parameters defined in the LLRP Specification. A Serialization module converts Message and Parameter binary objects to XML format, abstracting in this way the low level details of the LLRP interface.

Chapter 3

The Binding Framework

In this Chapter we describe the design and implementation of the XML binding framework for the LLRP Protocol. This framework serializes top-level objects according to the schema of the LLRP binary messages and parameters. It also validates the produced XML elements by checking that they conform to the LLRP schema. In the following sections we present the big picture of the system, and we describe the serialization procedure followed by a way to validate the results.

3.1 The Big Picture

In Figure 3-1 we see the big picture of the system. The LLRP client and the reader exchange messages in binary format. The binary messages are fed in the system (XML Serialization box in Figure 3-1). The system also takes an LLRP schema as an input (xsd file) and outputs an XML object (*XElement*)that corresponds to the encoded message or parameter. After the XML object is created, it is fed in the validation unit in order to make sure that it conforms to the LLRP schema.

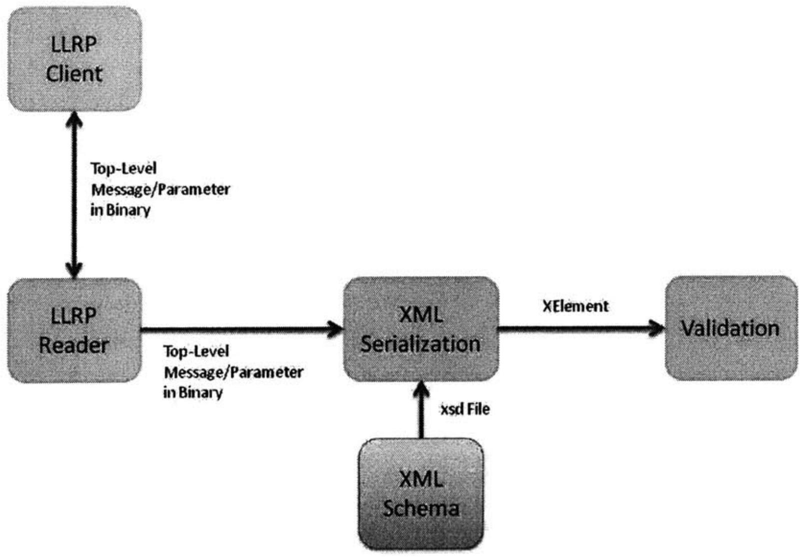

Figure 3-1: The Big Picture of the LLRP Binding Framework

3.2 LLRP Schema

For the purposes of our implementation we use the LLRP schema from the LLRP toolkit [8]. This is a W3C XML Schema that describes an XML format for a high-level, platform independent encoding of LLRP binary messages. XML-aware editors are able to use this schema to facilitate editing of XML messages. Instance documents can be archived in revision control systems, exchanged between developers, or used as part of a document-oriented API or web service for accessing LLRP readers. LLRP-XML instance documents or fragments are also ideal for quoting in tutorials and other documentation such as test plans.

According to the schema, every message is an XML root node with the name of the corresponding message as a header name. Every top-level message must have the *MessageID* as a required attribute and the *Version* as an optional attribute. The message header attribute group, *headerAttrs*, is shown in in Figure 3-2. The *MessageID* must be of type "unsigned integer" (a 32-bit unsigned integer, encoded using 4 bytes) and the *Version* must be of type "unsigned short integer" (a 16-

```
<xs:attributeGroup name="headerAttrs">
    <xs:attribute name="MessageID" type ="xs:unsignedInt"
                                   use="required"/>
    <xs:attribute name="Version" type="xs:unsignedShort" />

    <!-- overrides for negative testing only -->
    <xs:attribute name="Reserved" type ="xs:unsignedInt"/>
    <xs:attribute name="Length" type ="xs:unsignedShort"/>
    <xs:attribute name="TypeID" type ="xs:unsignedShort"/>
</xs:attributeGroup>
```

Figure 3-2: Message header attribute group

```
<xs:attributeGroup name="paramAttrs">

    <!-- overrides for negative testing only -->
    <xs:attribute name="Reserved" type ="xs:unsignedInt"/>
    <xs:attribute name="Length" type ="xs:unsignedShort"/>
    <xs:attribute name="TypeID" type ="xs:unsignedShort"/>
</xs:attributeGroup>
```

Figure 3-3: Parameter header attribute group

bit unsigned integer, encoded using 2 bytes). The attributes *Reserved*, *Length*, and *TypeID* are used for negative testing and are not included in our implementation.

In the same fashion, every high level parameter must have a parameter header attribute group which is called *paramAttrs*. Figure 3-3 shows the declaration of the parameter attributes.

All the messages are defined as top-level elements in the schema. An example of a top-level element is the *ADD_ROSPEC* message. The definition of this message in the schema is shown in Figure 3-4. As every message, this element has a header attribute group. The only element that this message contains is the *ROSpec* parameter. The definition of this parameter is shown in Figure 3-5. To define this parameter we need the values of the *ROSpecID* and *Priority* variables. Also we need the parameters *CurrentState*, *ROBoundarySpec*, *AISpec*, *RFSurveySpec*, which are also defined in the schema. The parameters that contain other parameters in their body appear in the schema as *complextypes*. The parameters that do not contain nested parameters

```
<xs:complexType name="ADD_ROSPEC">
    <xs:sequence>
        <xs:element name="ROSpec" type="llrp:ROSpec" />
    </xs:sequence>
    <xs:attributeGroup ref="llrp:headerAttrs"/>
</xs:complexType>
```

Figure 3-4: Definition of *ADD_ROSPEC* message

```
<xs:complexType name="ROSpec">
  <xs:sequence>
    <xs:element name="ROSpecID" type="xs:unsignedInt" />
    <xs:element name="Priority" type="xs:unsignedByte" />
    <xs:element name="CurrentState" type="llrp:ROSpecState" />
    <xs:element name="ROBoundarySpec" type="llrp:ROBoundarySpec" />
    <xs:choice maxOccurs="unbounded" >
      <xs:element name="AISpec" type="llrp:AISpec"/>
      <xs:element name="RFSurveySpec" type="llrp:RFSurveySpec"/>
      <xs:element name="Custom" type="llrp:Custom" minOccurs="0"
                                                   maxOccurs="unbounded" />
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
                                                        maxOccurs="unbounded"/>
    </xs:choice>
    <xs:element name="ROReportSpec" type="llrp:ROReportSpec" minOccurs="0" />
  </xs:sequence>
  <xs:attributeGroup ref="llrp:paramAttrs"/>
</xs:complexType>
```

Figure 3-5: Definition of *ROSpec* parameter

appear as *simpletypes*. Simpletypes usually contain an enumeration. An example of a simpletype parameter is *ROSpecState*, and its definition is shown in Figure 3-6. This parameter has an element of type string which can take one of the values "Disabled," "Active," or "Inactive."

3.3 Serialization

For every type of top-level message or parameter we have an *XElement* constructor class. All these classes contain a *Serialize* method which takes a *MessageEncoding* or *ParameterEncoding* as an argument. In this method, we retrieve the sub-elements of

```
<xs:simpleType name="ROSpecState">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Disabled"/>
        <xs:enumeration value="Active"/>
        <xs:enumeration value="Inactive"/>
    </xs:restriction>
</xs:simpleType>
```

Figure 3-6: Definition of *ROSpecState* parameter

the "encoding," and we create an XElement following the schema and using Xml.Linq.

As we saw in the big picture in Figure 3-1, our system takes an encoded message or parameter as an input. This *MessageEncoding* or *ParameterEncoding* object is passed to the *LLRPFactory* class which redirects this encoded message or parameter to the corresponding top-level constructor class. The high level design of the Serialization Process is shown in Figure 3-7.

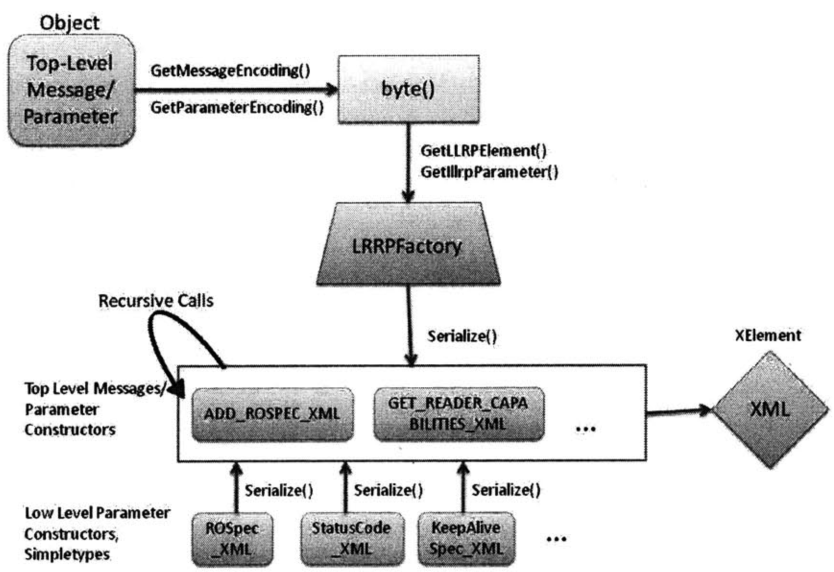

Figure 3-7: Serialization Process

One detail that makes the Serialization complicated is the fact that every message or parameter might contain nested messages or parameters. Our design treats these cases effectively as our top-level constructor classes call the serialize methods of each other repeatedly until all the levels of nested messages or parameters are serialized (Figure 3-7). All the Serialize methods return an object of type *XElement*. Consequently, the nested messages or parameters are constructed independently and the returned *XElement* object is appended in the appropriate XML node of the higher level message/parameter.

3.3.1 Top Level Messages

As we mentioned in the previous section, there is a separate constructor class for every type of message. The name of each constructor class is the name of the corresponding class concatenated with *"_XML"* (See Appendix A.2 for more details). As an example, we present the constructor class for an *ADD_ROSPEC* message.

As we saw in Figure 3-4, *ADD_ROSPEC* contains a *ROSpec* parameter. As we see in the code of Figure 3-8, we extract the *ROSpec* parameter and then give a call to the Serialize method of *ROSpec_XML* in order to get the nested *XElement roSpec*. This *roSpec* is then added to the root of our initial *XElement* which is the XML encoding for the *ADD_ROSPEC* message. The class diagram for the messages is shown in Appendix A.3.1.

```
public class ADD_ROSPEC_XML
    {
        public static XElement Serialize(MessageEncoding encoding)
        {
            XNamespace ns =
                "http://www.llrp.org/ltk/schema/core/encoding/xml/1.0";
            XElement root = new XElement(ns + "ADD_ROSPEC");

            ADD_ROSPEC message = new ADD_ROSPEC(encoding);
            ROSpec param = message.ROSpecParameter;

            XElement roSpec = ROSpec_XML.Serialize(
                    (TLVParameterEncoding)param.GetParameterEncoding());

            root.Add(roSpec);

            XAttribute id = new XAttribute("MessageID", encoding.MessageID);
            root.Add(id);

            Validator v = new Validator();
            v.Run(xe);

            return root;
        }
    }
```

Figure 3-8: The constructor class for an *ADD_ROSPEC* message

3.3.2 Parameters

The parameters are also constructed separately. There is a different constructor class for every type of parameter which contains a *Serialize(ParameterEncoding encoding)* method and returns an *XElement* object. In Figure 3-9 we see an example of constructor class (ROSpec_XML). The class diagram for the parameters is shown in Appendix A.3.2.

```
public class ROSpec_XML
{
    public static XElement Serialize(TLVParameterEncoding encoding)
    {
        XNamespace ns =
            "http://www.llrp.org/ltk/schema/core/encoding/xml/1.0";
        XElement root = new XElement(ns + "ROSpec");

        ROSpec param = new ROSpec(encoding);

        XElement roSpecID = new XElement(ns + "ROSpecID");
        roSpecID.Value = param.ROSpecID.ToString();

        XElement priority = new XElement(ns + "Priority");
        priority.Value = param.Priority.ToString();

        XElement currentState = ROSpecState_XML.Serialize(encoding);

        XElement roBoundarySpec = ROBoundarySpec_XML.Serialize(encoding);

        root.Add(roSpecID);
        root.Add(priority);
        root.Add(currentState);
        root.Add(roBoundarySpec);
        return root;
    }
}
```

Figure 3-9: The constructor class for a *ROSpec* parameter

3.3.3 SimpleTypes

The parameters that do not contain nested parameters appear as *simpletypes.* We already saw an example of the simpleType parameter *ROSpecState* in Figure 3-6. All the simpletypes have their own constructor classes. The *Serialize()* method of these classes works in the same fashion as the *Serialize()* method of the parameters. The only difference is that this method cannot be called outside the higher level constructor classes. This is because the simple types just add complexity to the messages or parameters but cannot exist by themselves as objects of a certain type. An example of the *ROSpecState* simpletype constructor class is shown in Figure 3-10. The class diagram for the simpletypes is shown in Appendix A.3.3.

```
private static XElement Serialize(TLVParameterEncoding encoding)
{
    XNamespace ns =
        "http://www.llrp.org/ltk/schema/core/encoding/xml/1.0";
    XElement root = new XElement(ns + "CurrentState");

    ROSpec param = new ROSpec(encoding);
    State data = (State) param.CurrentState;

    root.Value = data.ToString();
    return root;
}
```

Figure 3-10: The constructor class for a *ROSpecState* simpletype

3.4 Validation

The validation mechanism provides a way to test whether the XML objects created by our serialization methods are valid. The *validator* contains a *Run()* method that takes an *XElement* object as an argument. The validator takes an xsd file with the LLRP schema and checks whether the input conforms to the schema. The code for the validation unit is shown in Appendix B. The validator also serves for debugging purposes as it can validate any complex type of message or parameter.

3.5 Testing

3.5.1 Message Tests

All messages are tested in individual unit tests. In these units tests we create new instances of messages, assign them random values, encode them in binary, and then serialize them to XML. The test passes if the resulted *XElement* that corresponds to the message passes the validation procedure that we described earlier. Figure 3-11 illustrates a template code for testing the *ADD_ROSPEC_XML* constructor class.

```
class ADD_ROSPECTest
{
    public boolean void Run()
    {
        ADD_ROSPEC myMsg = new ADD_ROSPEC();
        MessageEncoding me = myMsg.GetMessageEncoding();

        XElement xe = LLRPFactory.GetLLRPElement(me);

        Validator v = new Validator();
        return(v.Run(xe));
    }
}
```

Figure 3-11: A simple test case for *ADD_ROSPECTest* simpletype

As we see in Figure 3-11, a new instance of type *ADD_ROSPEC* is created. The message is then encoded and passed to the *LLRPFactory* module. The XML object is stored in *xe* and the validator runs against it. The test passes if the validator returns true. The XML view of *xe* is shown in Figure 3-12.

3.5.2 Test Manager

We input all the test cases in a test Module and run tests against all message types. The test manager reports if there are any types of messages with corresponding XML objects that do not conform to the schema.

Testing of the individual parameters is not necessary because they are contained

```xml
<ADD_ROSPEC Version="1" MessageID="1">
  <ROSpec>
    <ROSpecID>1</ROSpecID>
    <Priority>0</Priority>
    <CurrentState>Disabled</CurrentState>
    <ROBoundarySpec>
      </ROSpecStartTrigger>
      <ROSpecStopTrigger>
        <ROSpecStopTriggerType>Duration</ROSpecStopTriggerType>
        <DurationTriggerValue>0</DurationTriggerValue>
      </ROSpecStopTrigger>
    </ROBoundarySpec>
    <AISpec>
      <AntennaIDs>1 2 3 4</AntennaIDs>
      <AISpecStopTrigger>
        <AISpecStopTriggerType>Null</AISpecStopTriggerType>
        <DurationTrigger>0</DurationTrigger>
      </AISpecStopTrigger>
      <InventoryParameterSpec>
        <InventoryParameterSpecID>1</InventoryParameterSpecID>
        <ProtocolID>EPCGlobalClass1Gen2</ProtocolID>
      </InventoryParameterSpec>
    </AISpec>
  </ROSpec>
</ADD_ROSPEC>
```

Figure 3-12: The XML view of *ADD_ROSPEC* in messages and are being tested when the message is being tested. Note that the validation procedure is being processed at the end of the serialization process of any parameter or message, and so our testing is restricted to high-level testing, in the sense that top-level elements are being tested.

Chapter 4

Discussion and Future Work

4.1 Discussion

Developing the XML Binding Framework for the Low Level Reader Protocol (LLRP) involved crucial design decisions. Below are the advantages of handling the received data in XML:

Ease of Implementation

The object model allows for an easy to develop and robust Framework. Using the abstraction of the LLRP Library [9], we managed to create a mechanism that decrypts the binary encoded messages easily and produces an XML object. Using *Xml.Linq* Library we managed to serialize nested messages and parameters in an efficient way. Every nested parameter is just another *XElement* node which is inserted in the appropriate node of the larger *XElement* tree structure.

Advantages of XML Serializing

The structured nature of an LLRP message or parameter allows for an easy XML serialization process. When parsing XML, less expertise are needed, and, as such, broader adoption is enabled. The use of *XPath* or *XQuery* would also allow fast searching through an XML encoded Message and could allow Messages to be pre-processed before converting the XML into the object model. Converting a message at the reader's side into XML allows for a better view of the object and easy to modify data for testing/debugging purposes.

Error handling

In our implementation, we made sure that all the mandatory fields are being checked, and all the levels of nested parameters are validated against the schema. "Bad" encodings are treated in a user friendly manner, and the validation describes the exact misinterpretation of the schema.

4.2 Future Work

4.2.1 Optional Fields

Some of the optional fields are not included in the parameter constructor classes. These are the parameters under the *AirProtocolSpcic* namespace. The implementation of these optional fields will complete our implementation.

4.2.2 Graphical User Interface

The goal of this implementation was simply to illustrate how the LLRP library can be used to develop a Serialization Framework. As a future extension a GUI could provide us with a better view of the messages being passed from the tags to the readers. Additional functionality could include handling multiple Readers and providing ways to automate Reader control. Readers can be controlled to send status reports either periodically or triggered by some event. These reports can include the XML objects out of the messages, the readers handle. The Client could provide logic to automatically process Reader responses and perform different actions depending on Reader status.

4.2.3 Auto-generated Message & Parameter Constructor Classes

Using the LLRP schema it would be possible to create a module that would automatically generate all Message and Parameter classes, since the class structure is similar across Messages and Parameters. This would make the implementation of an LLRP library simpler and more robust. Possible future changes in the LLRP specification would be reflected in the schema and would not require any changes in the code. Moreover, the auto-generation module could be modified to provide Parameter Range checking and improved error checking without the need to tediously go through each Message and Parameter class.

Chapter 5

Conclusions

This Master of Engineering Thesis describes the design, implementation and testing of an XML binding framework for the Low Level Reader Protocol (LLRP). This framework serializes top-level objects according to the schema of the LLRP binary messages and parameters. It also validates the produced XML elements by checking that they conform to the LLRP schema. The use of this framework provides a data serialization mechanism at the reader's side and allows for easy and efficient data update when simulating the RFID Network. As RFID technology continues to be more widely adopted, it becomes imperative to standardize the way information is stored at the reader's side. A standard interface will allow RFID systems to be designed without dependence on vendor proprietary interfaces and will simplify their implementation. It is envisioned that LLRP will soon be adopted as the standard Reader - Client interface.

The implementation described in this Thesis was oriented towards reducing the complexity of LLRP, which primarily lies in the binary nested format used for representing LLRP Messages and Parameters. Our system uses the nested object model that is used to represent all the Messages an Parameters defined in the LLRP Specification. A Serialization module converts Message and Parameter binary objects to XML format, abstracting in this way the low level details of the LLRP interface. This *LLRP_XML* library can be a useful toolkit in developing LLRP applications, without requiring the investment of too much time and effort in dealing with data handling as all messages and parameters will be converted to XML and then edited/updated in the XML environment.

Appendix A

Design Details

A.1 Casing Conventions

The following conventions are used for naming the various classes of the LLRP library:

ALL_CAPS_UNDERSCORE type is used for LLRP message names (Top-Level Elements), e.g. *GET_READER_CAPABILITIES*

Camel casing is used for LLRP Parameter and data field names. e.g. ROSpec Interfaces use camel casing and are prefixed with an "I," e.g. *IConfigGetParameter.*

In this Binding Framework we name all the classes that construct the XML Serialization with the name of the class of the corresponding object appended with "_XML."

A.2 Namespaces

LLRP_XML.Messages.AccessOperation

Messages that query Reader Capabilities.

- GET_READER_CAPABILITIES
- GET_READER_CAPABILITIES_RESPONSE

LLRP_XML.Messages.ReaderOperation

Messages that control the Readers air protocol inventory and RF operations.

- ADD_ROSPEC
- ADD_ROSPEC_RESPONSE
- DELETE_ROSPEC
- DELETE_ROSPEC_RESPONSE
- START_ROSPEC
- START_ROSPEC_RESPONSE
- STOP_ROSPEC
- STOP_ROSPEC_RESPONSE
- ENABLE_ROSPEC
- ENABLE_ROSPEC_RESPONSE
- DISABLE_ROSPEC
- DISABLE_ROSPEC_RESPONSE
- GET_ROSPECS
- GET_ROSPECS_RESPONSE

LLRP_XML.Messages.AccessOperation

Messages that control the tag access operations performed by the Reader.

- ADD_ACCESSSPEC
- ADD_ACCESSSPEC_RESPONSE
- DELETE_ACCESSPEC

- DELETE_ACCESSSPEC_RESPONSE
- ENABLE_ACCESSSPEC
- ENABLE_ACCESSSPEC_RESPONSE
- DISABLE_ROSPEC
- DISABLE_ROSPEC_RESPONSE
- GET_ACCESSSPECS
- GET_ACCESSSPECS_RESPONSE

LLRP_XML.Messages.ReaderDeviceConfiguration

Messages that query/set Reader configuration, and close LLRP connection.

- GET_READER_CONFIG
- GET_READER_CONFIG_RESPONSE
- SET_READER_CONFIG
- SET_READER_CONFIG_RESPONSE
- CLOSE_CONNECTION
- CLOSE_CONNECTION_RESPONSE
- DISABLE_ROSPEC
- DISABLE_ROSPEC_RESPONSE
- GET_ACCESSSPECS
- GET_ACCESSSPECS_RESPONSE

LLRP_XML.Messages.ReportsNotificationsKeepalives

Messages that carry different reports from the Reader to the Client. Reports include Reader device status, tag data, RF analysis report.

- GET_REPORT
- RO_ACCESS_REPORT
- READER_EVENT_NOTIFICATION
- KEEPALIVE
- KEEPALIVE_ACK
- ENABLE_EVENTS_AND_REPORTS

LLRP_XML.Messages.CustomExtension

Messages that contain vendor defined content.

- CUSTOM_MESSAGE

LLRP_XML.Messages.Errors

Generic error messages.

- ERROR_MESSAGE

LLRP_XML.Parameters.ReaderOperation

Parameters for Messages that control the Readers air protocol Inventory and RF operations.

LLRP_XML.Parameters.AccessOperation

Parameters for Messages that control the tag access operations performed by the Reader.

LLRP_XML.Parameters.ReaderDeviceConfiguration

Parameters for Messages that query/set Reader configuration, and close LLRP connection.

LLRP_XML.Parameters.ReportsNotificationsKeepalives

Parameters for Messages that carry different reports from the Reader to the Client.

LLRP_XML.Parameters.ReportsNotificationsKeepalives.ReaderEventNotificationData

Parameters for Messages that carry different event notification reports from the Reader to the Client.

LLRP_XML.Parameters.ReportsNotificationsKeepalives.TagReportData

Parameters for Messages that carry different tag data reports from the Reader to the Client.

LLRP_XML.Parameters.ReaderDeviceCapabilities

Parameters for Messages that query Reader capabilities.

LLRP_XML.Parameters.ReaderDeviceCapabilities.GeneralDeviceCapabilities

Parameters for Messages that query general device capabilities of a Reader.

LLRP_XML.Parameters.ReaderDeviceCapabilities.RegulatoryCapabilities

Parameters for Messages that query regulatory device capabilities of a Reader.

LLRP_XML.Parameters.General

General Parameters.

LLRP_XML.Parameters.AirProtocolSpecific.C1G2

Parameters for the C1G2 air protocol.

LLRP_XML.Parameters.CustomExtension

Parameters for Messages that contain vendor defined content.

LLRP_XML.Parameters.Errors

Parameters for generic error Messages

A.3 Class Diagrams

A.3.1 Messages

Figure A-1: Class Diagram for the Message-generator classes

A.3.2 Parameters

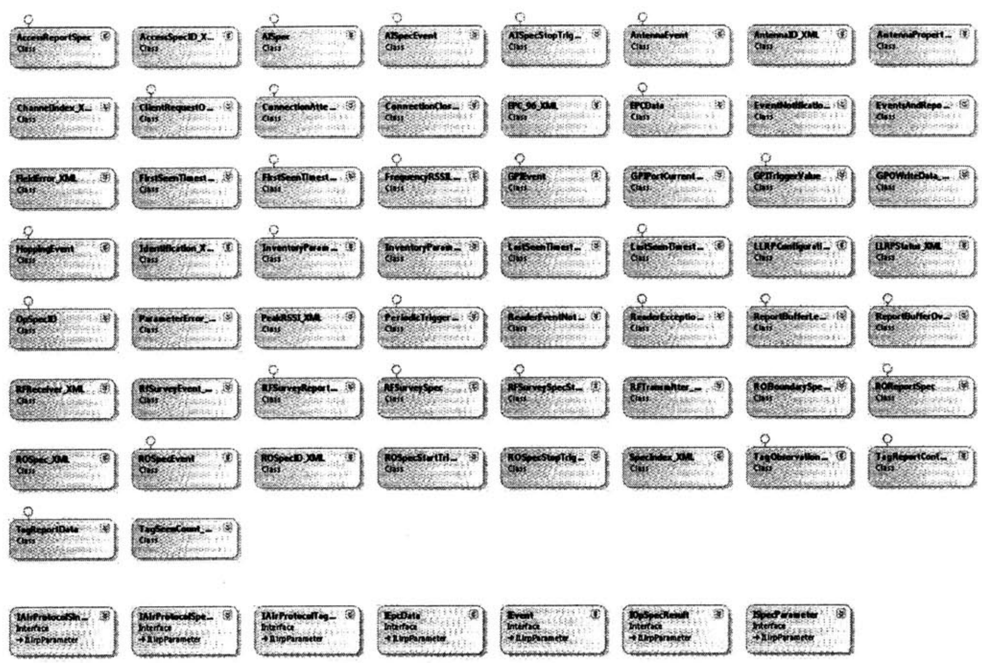

Figure A-2: Class Diagram for the Parameter-generator classes

A.3.3 Simpletypes

Figure A-3: Class Diagram for the *SimpleTypes*

A.3.4 Tests

Figure A-4: Class Diagram for the Test Classes

Appendix B

Validation Class

Validator.cs

```
using System; using System.Collections.Generic;
using System.Text;
using System.Xml.Linq;
using System.Xml.Schema;
using System.IO;
using System.Xml;
using System.Web;

namespace LLRP_XML.XML_Factory {
    class Validator
    {
        private bool error = false;
        private bool warning = false;
        private string help = string.Empty;
        private List<string> helpLinks = new List<string>();
        private string message = string.Empty;
        private string messageHtml = string.Empty;
        private DateTime testEndTime = System.DateTime.Now;
```

```
private DateTime testStartTime = System.DateTime.Now;
private string xmlTagAndPayload = string.Empty;
private string xmlTagAndPayloadInHtmlForm = string.Empty;
private string xmlTagName = string.Empty;
private string schemaPath ="C:\\Users\\net\\Documents\\Visual Studio
                           2008\\Dimpoul\\LTK\\Definitions\\Core\\";

/// <summary>
/// This test passes if and only if the document is
/// valid as an EPCISDocument according to the schema
/// that is specified in Section 3.2 (llrp-1x0.xsd)
/// </summary>
public void Run(XElement epcisDocument)
{
    try
    {
        testStartTime = System.DateTime.Now;
        help = "EPCISDocument schema validation failed. " +
                "This test passes if and only if the document is " +
                "valid either as an EPCISDocument according to " +
                "the schema that is specified in Section 9.5 of " +
                "the EPCIS 1.0.1 spec, or as an EPCISQueryDocument " +
                "according to the schema specified in Section 11.1 " +
                "of the EPCIS 1.0.1 spec.";
        helpLinks.Add("http://www.epcglobalinc.org/");
        helpLinks.Add("http://www.epcglobalinc.org/standards/epcis");
        message = "EPCISDocument schema validation failed. ";

        if (epcisDocument == null) return;
```

```
            XmlSchemaSet epcisSchemaSet = new XmlSchemaSet();
            epcisSchemaSet.Add(schemaPath + "llrpdef.xsd");
            epcisSchemaSet.Add(schemaPath + "llrp-1x0.xsd");

            SchemaValidation schemaValidation =
                new SchemaValidation(epcisSchemaSet);
            if (schemaValidation.Validate(epcisDocument.ToString()) == false)
            {
                System.Console.WriteLine("Validation did not succeed");
                error = true;

                messageHtml = message + schemaValidation.ErrorMsg + ". ";
                xmlTagName = "EPCISDocument";
            }
            else
            {
                System.Console.WriteLine("Validation passed");
            }

            testEndTime = System.DateTime.Now;
        }
        catch (Exception e)
        {
            error = true;
            System.Console.WriteLine("An exception occured");
            messageHtml = e.Message;
        }
    }

    /// <summary>
```

```
/// The name of the tag being validated
/// </summary>
public string XmlTagName { get { return xmlTagName; } }

/// <summary>
/// The name of the tag being validated and the payload
/// </summary>
public string XmlTagAndPayload { get { return xmlTagAndPayload; } }

/// <summary>
/// The name of the tag being validated and the payload
///  - formatted in Html
/// </summary>
public string XmlTagAndPayloadInHtmlForm
{ get { return xmlTagAndPayloadInHtmlForm; } }

/// <summary>
/// The message formatted in Html
/// </summary>
public string MessageHtml { get { return messageHtml; } }
/// <summary>
/// Error value is true if validation produces error
/// </summary>
public bool Error { get { return error; } }

/// <summary>
/// Warning value is true if validation produces warning
/// </summary>
public bool Warning { get { return warning; } }
```

```
/// <summary>
/// Clock time at test start
/// </summary>
public DateTime TestStartTime { get { return testStartTime; } }

/// <summary>
/// Clock time at test end
/// </summary>
public DateTime TestEndTime { get { return testEndTime; } }

/// <summary>
/// Validation message
/// </summary>
public string Message { get { return message; } }

/// <summary>
/// Help message
/// </summary>
public string Help { get { return help; } }

/// <summary>
/// Links to help resources
/// </summary>
public List<string> HelpLinks { get { return helpLinks; } }

private class SchemaValidation
{
    private XmlSchemaSet epcisSchemaSet;
    private StringReader epcisEventDataReader;
```

```
private XmlReaderSettings validationSettings;
private bool errorsFound = false;
private StringBuilder errorDetail = new StringBuilder();
private string errorMsg = string.Empty;

public string ErrorMsg { get { return errorMsg; } }

public SchemaValidation(XmlSchemaSet schemaSet)
{ epcisSchemaSet = schemaSet; }

public bool Validate(string epcisEventData)
{
    epcisEventDataReader = new StringReader(epcisEventData);
    validationSettings = new XmlReaderSettings();
    validationSettings.Schemas.Add(epcisSchemaSet);
    validationSettings.ValidationType = ValidationType.Schema;
    validationSettings.ValidationFlags
        |= XmlSchemaValidationFlags.ProcessInlineSchema;
    validationSettings.ValidationFlags
        |= XmlSchemaValidationFlags.ReportValidationWarnings;
    validationSettings.ValidationEventHandler +=
        new ValidationEventHandler(ValidationCallback);

    XmlDocument docXml = new XmlDocument();
    XmlReader validator =
        XmlReader.Create(epcisEventDataReader, validationSettings);
    docXml.Load(validator);

    validator.Close();
```

```
                if (errorsFound)
                {
                    System.Console.WriteLine("Errors Found");
                    errorMsg = errorDetail.ToString();
                    return false;
                }
                else
                    return true;
            }

            private void ValidationCallback(object sender,
                                  ValidationEventArgs args)
            {
                string singleError = String.Empty;

                if (args.Severity == XmlSeverityType.Warning)
                    singleError = "WARNING: ";
                else if (args.Severity == XmlSeverityType.Error)
                {
                    singleError = "ERROR: ";
                    errorsFound = true;
                }

                singleError += args.Message;
                errorDetail.AppendLine(singleError);
                Console.WriteLine(singleError);
            }
        }
    }
}
```

Bibliography

[1] EPCGlobal, *The EPCGlobal Architecture Framework.*, EPC Global Standard Specification, July 2006.

[2] EPCGlobal, *The application Level Events (ALE) Specification*, Version 1.0 EPC Global Standard Specification, 2005.

[3] EPCGlobal, *Object Naming Service (ONS)*, Version 1.0 EPC Global Standard Specification, 2005.

[4] EPCGlobal, *EPC Information Services (EPCIS)*, Version 1.0 EPC Global Standard Specification, April 2007.

[5] EPCGlobal, *Low Level Reader Protocol (LLRP)*, Version 1.0 EPC Global Standard Specification, April 2007.

[6] Ane Fabo, *Design and Implementation of an Application for Management and Control of RFID readers*, May 2008

[7] Sam Polniak, *The RFID Case Study Book*

[8] John R. Hogerhuis, *http://sourceforge.net/projects/llrp-toolkit/*, LLRP Toolkit, 2007

[9] Fivos Constantinou, *An Object-Oriented Implementation of a Low Level Reader Protocol (LLRP) Library*, MIT June 2007