SmartRF: A Flexible and Light-weight RFID Middleware

Anirudh Ghayal

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

June, 2008

SmartRF: A Flexible and Light-weight RFID Middleware

A Thesis Report Submitted

in Partial Fulfillment of the Requirement

for the Degree of

Master of Technology

by

Anirudh Ghayal

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

June, 2008

Certificate

This is to certify that the work contained in the thesis titled *"SmartRF: A Flexible and Light-weight RFID Middleware"*, by *Anirudh Ghayal*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

June, 2008 ——————————————————————

(Dr. Rajat Moona)

Department of Computer Science & Engineering,

Indian Institute of Technology Kanpur

——————————————————————

(Dr. A R Harish)

Department of Electrical Engineering,

Indian Institute of Technology Kanpur

Abstract

Radio frequency based Identification (RFID) is expected to be deployed in a major way in the near future. The major issues for such a deployment are in the design of a robust and flexible software system to interface various applications to the RFID readers. There are few existing RFID software systems, most of which are proprietary and the others are under development. The proprietary RFID software solutions are costly, bulky, non-portable and heavily dependent on the support software.

In this work, we present SmartRF, an open-source RFID middleware which is flexible, simple and scalable. SmartRF allows us to interface RFID tags and readers to multiple applications in a technology-neutral (protocols, air-interface, etc.) manner. The object-oriented and layered design of SmartRF allows development and integration of new features with little effort. The middleware provides the application a flexibility to interact with one or more readers or even part of a reader. SmartRF provides the application developer with a simple and hardware-independent set of APIs to access and configure the hardware. It supports dynamic joining and dis-joining of applications and hardware thus, providing flexibility to the system.

Two RFID systems – electronic file tracking and postal bag tracking, were developed with the help of SmartRF and a supporting application framework.

Acknowledgments

I wish to thank my thesis advisers, Dr. Rajat Moona and Dr. A R Harish for all their support and guidance. Their encouragement and enthusiasm has been my primary source of inspiration and motivation. The brain-storming weekly discussions on the different approaches involved in the middleware design and implementation have been a valuable experience.

I thank Prof. Veena Bansal for her invaluable advice in my thesis. Her expertize in understanding and analyzing the problems involved in the real world deployment of a RFID system have been immensely helpful in developing a user friendly system.

I thank Mohd. Zuber Khan, my classmate, for being actively involved in all phases of my thesis. Without his help and support, the thesis could not have progressed. The application framework and middleware GUI developed by him have been extremely helpful in the development of the RFID system. I am also thankful to all my MTech friends for constantly encouraging and supporting me throughout my stay here at IIT Kanpur.

I thank the entire Computer Science Department for making such an enjoyable place to work. The support of the administrative and technical staff helped in easy access of resources as and when needed.

Finally, I am grateful to my parents and my sister for all their love and support. Their constant affection and encouragement has helped me achieve my goals.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 3
- 1.2 RFID System Overview . . . 4
  - 1.2.1 RFID System Components . . . 5
- 1.3 Related Work . . . 8
  - 1.3.1 RFID Middleware . . . 9
  - 1.3.2 RFID Applications . . . 11
- 1.4 Organization of the Report . . . 12

2 RFID Middleware **13**
- 2.1 Middleware Components . . . 14
  - 2.1.1 Reader Interface . . . 14
  - 2.1.2 Data Processor and Storage . . . 15
  - 2.1.3 Application Interface . . . 15
  - 2.1.4 Middleware Management . . . 16
- 2.2 Middleware Design Issues . . . 17

3 SmartRF – The RFID Middleware **19**
- 3.1 Hardware Abstraction Layer (HAL) . . . 20
- 3.2 Event and Data Management Layer (EDML) . . . 22
- 3.3 Application Abstraction Layer (AAL) . . . 23
  - 3.3.1 Data Streams . . . 23
  - 3.3.2 Channels . . . 25

4 SmartRF Implementation **28**
- 4.1 Implementation Model . . . 28
  - 4.1.1 Client-Server Model . . . 28
  - 4.1.2 Multi-threaded Architecture . . . 29
- 4.2 HAL Implementation . . . 31
  - 4.2.1 HAL Driver Functions . . . 33
    - 4.2.1.1 Open Reader . . . 33
    - 4.2.1.2 Close Reader . . . 33
    - 4.2.1.3 Configure Reader . . . 34
    - 4.2.1.4 Start Read Operation . . . 34
    - 4.2.1.5 Get Data . . . 35
    - 4.2.1.6 Stop Reader Read . . . 35

4.2.1.7 Number of tags available . . . 35
4.2.1.8 Write Data . . . 35
4.2.1.9 Select Antenna . . . 36
4.2.1.10 Set Association . . . 36
4.2.1.11 Check Reader Connectivity . . . 37
4.3 EDML Implementation . . . 37
4.4 AAL Implementation . . . 39
4.4.1 AAL API Specification . . . 40
4.4.1.1 Connect to Middleware . . . 40
4.4.1.2 Get Reader List . . . 41
4.4.1.3 Create Channel . . . 41
4.4.1.4 Change Channel Configuration . . . 42
4.4.1.5 Get Channel Configuration . . . 42
4.4.1.6 Destroy Channel . . . 43
4.4.1.7 Read Data . . . 43
4.4.1.8 Write data . . . 43
4.4.1.9 Exit Application . . . 44
4.5 Initialization of SmartRF . . . 44

5 SmartRF Applications 49
5.1 Application Types . . . 49
5.2 Application development on SmartRF . . . 50
5.3 Postal Bag Tracking Application . . . 53

6 Performance and Results 57
6.1 Setup Environment . . . 57
6.2 Performance Parameters . . . 58
6.3 Test Specifications . . . 58
6.4 Performance Results . . . 59
6.4.1 Initialization Time . . . 59
6.4.2 Response Time . . . 59
6.4.3 Runtime and Peak Memory Usage . . . 60

7 Conclusion 62

A Hardware Abstraction Layer (HAL) APIs of SmartRF 70
A.1 Introduction . . . 70
A.2 System Architecture . . . 71
A.3 API Specification . . . 72
A.3.1 Open Reader . . . 72
A.3.2 Close Reader . . . 72
A.3.3 Configure Reader . . . 73
A.3.4 Start Read Operation . . . 74
A.3.5 Get Read Data . . . 75
A.3.6 Stop Reader Read . . . 76
A.3.7 Number of tags available . . . 77

A.3.8 Write Data 77
A.3.9 Select Antenna 78
A.3.10 Set Associations 79
A.3.11 Check Reader Connectivity 80

B Application Abstraction Layer (AAL) APIs of SmartRF 81
B.1 Introduction 81
B.2 System Architecture 82
B.3 API Specification 83
B.3.1 Connect to the middleware 83
B.3.2 Get reader list 83
B.3.3 Create Channel 85
B.3.4 Change channel configuration 86
B.3.5 Get channel configuration 87
B.3.6 Destroy Channel 88
B.3.7 Read data 89
B.3.8 Write data 89
B.3.9 Exit application 90

List of Tables

4.1 Channel Parameters 42
4.2 Reader port parameters 45

6.1 Initialization time in milliseconds of SmartRF 59
6.2 Response time in milliseconds of SmartRF 60

List of Figures

1.1 Example RFID System 4
1.2 RFID System Components 6

2.1 RFID Middleware Components 15

3.1 SmartRF System Design 21
3.2 Data Streams, Channels and Association 24

4.1 Client-Server Model of SmartRF 29
4.2 Multi-threaded Architecture of SmartRF 30
4.3 HAL Device Driver 32
4.4 SmartRF – Application Interaction 40
4.5 SmartRF Initial Configuration File 47

5.1 RFID Gate 53
5.2 Portal for the Bag Tacking Information 55
5.3 Comparison between Tag A and Tag B 56

6.1 Runtime memory usage of SmartRF 61

A.1 RFID System Architecture (hardware) 71

B.1 RFID System Architecture (applications) 82

Chapter 1

Introduction

Radio Frequency based Identification, known as RFID, is being described as "the next big thing in technology". The sudden spur in this technology is due to huge corporate[30] and government[9] investments. These investments are also responsible for the increase in the research and development in this technology. RFID is a technology which concerns auto-identification (Auto-ID). There exist, a number of auto-identification systems like the barcode based systems, optical recognition systems (OCR) which are used extensively.

The barcode based identification systems use a binary code comprising field of bars and gaps, arranged in a parallel configuration. This sequence, made by narrow and wide bars is interpreted numerically and alphanumerically by analyzing the reflected laser beam on the bar gaps. The interpreted value obtained, specifies a unique code which is used to identify the object. The problem in this system is that of the need to expose the barcode manually to the laser beam. The barcode needs to be aligned to be read by the laser scanner. Inspite of these drawbacks, the barcode systems are one of the most widely used auto-ID systems. OCR based systems consists of optical machine readers which are used in auto-identification. These readers recognize alphanumeric codes which are placed on the objects, to uniquely identify the objects. The disadvantage of this system is in the cost of operation and the complexity of the OCR readers[12].

RFID systems[16, 34] are relatively new entrants to the auto-ID systems and have become very popular recently. Though, the RFID based systems have existed since 1960[12], the reason for their recent popularity is the development in the field of semiconductors which has led to the decrease in cost of hardware. This has made it possible to develop cheap[15] and reliable RFID hardware. RFID systems are being used in variety of areas ranging from inventory tracking to access control systems[7, 8, 33]. The ability to identify items which are not in the line of sight has given this technology an edge over other auto-ID systems. The integration of this technology in various business domains have reduced losses[9] and improved the overall efficiency of the working system. For example, losses in the supply chain management systems have been huge due to mismanagement[37, 22]. RFID integration[4, 22] in such systems leads to improved visibility in various stages of supply-chaining, thus increasing the efficiency of the system.

In RFID systems, the identifications data is stored on an electronic data-carrying devices known as tags[12]. These tags are placed on the objects which are then uniquely identified. The identification takes place when these tags move in the vicinity of the RFID readers. The RFID readers are devices which communicate and access data from the RFID tags using radio waves. The readers physically transmit and receive data through radio waves using the antennas connected to them. The identification data read by the readers is then processed by the software system, known as the middleware. This data then can be accessed by various RFID applications by communicating with the middleware. Thus, the middleware acts like a server, which services a number of applications on one side and connects to the hardware on the other side. The major advantages of using RFID as an auto-ID system are the following.

- RFID readers do not need a line of sight to access data from the RFID tags.
- RFID systems can read data over varied distances. The range varies from few centimeters to few hundred meters.

- RFID readers can interrogate, or read, RFID tags much faster.
- RFID systems can read and write different sizes of data from / to the tag, based on the type of tag.
- RFID systems can read tags in harsh environments, without any human interference.

There has been a lot of research[28, 15] in the development of efficient, durable and cheap RFID tags and readers. But very little has been done in the development of the middleware and the application framework. There is a need to develop a simple and robust middleware framework which allows easy development and deployment of applications.

1.1 Problem Statement

The objective of this thesis is to design and develop a simple, light-weight and flexible RFID middleware. The middleware must provide the applications with a device-neutral interface to communicate with the hardware. The middleware design must support simultaneous communication of multiple applications with the RFID hardware. The applications should be provided with a set of function calls to access the functionality of the middleware. Similarly, the design must provide an API for the hardware to be accessible from the middleware. The middleware must provide all data processing capabilities like filtering, grouping and duplicate data removal.

An objective of this thesis is to develop the middleware as an open-source software which should allow integration of new features with little effort. Therefore the middleware is desired to carry a modular and layered design. An important consideration in the development and deployment of the middleware is its dependency on other support software like database, application servers etc. The run time dependency on these support software should be minimal making the middleware portable and light-weight.

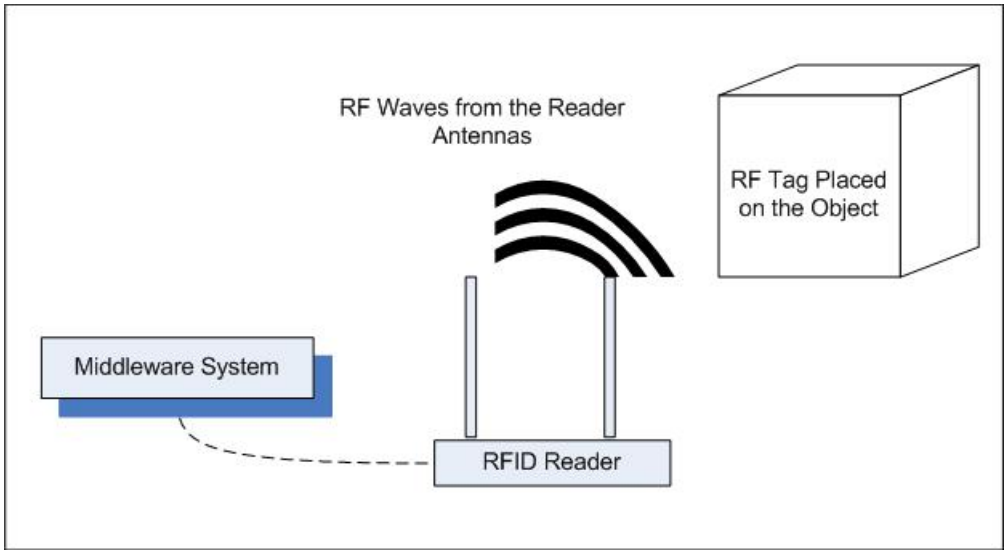


Figure 1.1: Example RFID System

Finally, an objective of this thesis had been to show the utility of the middleware by developing an application – The Postal Bag Tracking System.

1.2 RFID System Overview

The basic RFID system (figure 1.1) consists of tags which are placed on the objects to be identified. These tags are identified by the RFID readers, when the object passes through the antennas connected to the reader. The data read by the reader, is then processed by the middleware and presented to the application. Thus, the data flows from the tags to the readers, then to the middleware and finally to the application. At each of these stages the data is stored and processed. This capture, processing and handling of the data, hardware and software is a part of the RFID system.

RFID systems are being use extensively in various domains. The requirements from these systems vary depending on the type of application for which the system is deployed. For example, a typical tag and ship application for a consumer good manufacturer would mainly look into the physical needs of the hardware like the placement of the physical readers in appropriate position for maximal tag reads,

efficient tag placement on objects. In such applications, there is very minimal report generation and processing of data. Whereas, applications like asset tracking would need minute-by-minute update of data, which makes the software involved, the critical part of the complete setup. An RFID system requires to take care of different needs of various applications. The RFID systems will continue to evolve to meet wide spectrum of needs, requiring various architectures. Hence, it might not be possible to define a general RFID architecture but, there are certain functions and capabilities which must be supported by all RFID systems.

1. The ability to encode the RFID tags with an unique ID to identify the object.
2. The ability of the RFID readers to identify the tags and the ability of the RFID system to track tags as they move from one location to another (i.e reader to reader).
3. The ability to access and process the data read by the RFID readers.
4. The ability to integrate the system to support enterprise applications.
5. The ability to share information between various applications.

An RFID system which supports all these basic capabilities can be easily extended to support multiple kinds of applications.

1.2.1 RFID System Components

An RFID system generally consists of four major components[16], of which some are a part of the hardware system and the others constitute the software system, as shown in figure 1.2.

RFID Tags are hardware electronic components that carry data. They are generally placed or embedded on the objects which are to be uniquely identified. RFID tags are accessed by the RFID readers using the radio waves. The tags are classified

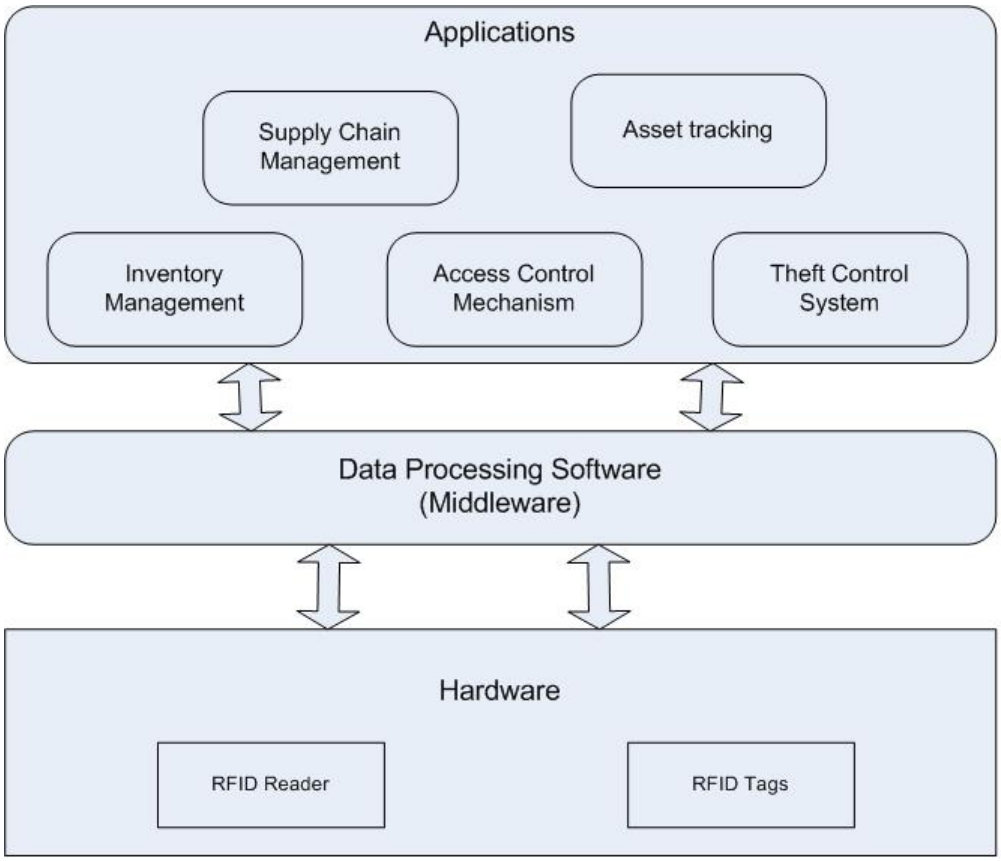


Figure 1.2: RFID System Components

based on a number of characteristic features[16] like power, air-interface, packaging, protocols, storage capacity, form factor and cost.

Tags may be classified as passive or active tags. Passive tags are the ones which do not have a power source of their own. They are powered by the electro-magnetic radio waves from the reader and generally have smaller range of communication. Active tags on the other hand, have an inbuilt power source which allows them to transmit radio waves independently without the support of the reader. Active tags are costlier compared to the passive tags, but are more powerful and have a longer range.

Tags provide storage capacity ranging from a few bits to a few hundred bytes. Tags are further divided based on their access capability. The read-only tags are the ones which do not permit the modification of data and carry a permanent ID. The read-write tags are the ones which permit multiple read and write operations. A tag generally has two different kinds of memories, one for the unique ID (UID) and other for the user data. The UID memory varies from 4 byte[12] to 12 bytes depending upon the RFID protocol. On the other hand, the user data memory is optional and comes in 0 to few kilo bits capacity.

The tags can be communicated with only over the the radio frequency supported by them. HF, UHF and Microwave are some of the standard frequencies supported by the tags. A tag can be accessed only by the reader which transits and receives radio waves in the same frequency range. For example - EPC Gen 2 tags are designed to support UHF frequency range (865 MHz to 915 MHz), these tags can be read only by the UHF readers having the same or a subset of the tag's frequency. Some of the companies which manufacture tags are - Alien[10], UPM RAFLATAC[32] etc.

The RFID readers are hardware components of the RFID system which communicate with the tags using radio waves. The readers transmit these waves via the antennas connected to them. Readers are connected to computer systems running the suitable reader access software. The readers are characterized by a number of features such as air interface, protocol, host-side interface etc.

As in the case of tags, the readers are also designed to support a specific frequency range. RFID tags and the corresponding readers use a specific protocol for data communication. These protocols define how the data is organized in a tag and the access mechanism to be used by the reader to communicate and access data from the tag. RFID readers are generally developed to support a single protocol, but they may also support multiple protocols. Some examples of protocols are EPC Gen2[14], ISO 15693[3], ISO14443A[2] etc.

The readers are connected to the computer system using various communication interface like USB, Serial Port, Ethernet port. The computer then uses this interface to communicate with the reader. All the commands and data which flow from the computer to the reader and vice versa follow the same communication interface.

The middleware is the data processing software that manages data and control flow between the RFID readers and applications. It interfaces with the RFID hardware on one end and with the RFID application software on the other. A middleware standardizes the way of dealing with the flood of tag information from the readers. It also hides the physical details of the system from the applications – so that applications can be written in a device-neutral manner. The middleware performs data processing operations like filtering, grouping of the data based on the configuration done by the application. The middleware thus plays a very important role in managing the complete RFID system.

Applications are the software components which configure the middleware and logically interpret the data. Applications act as the end-user interface to the complete RFID system. RFID applications serve various purposes ranging from article tracking to theft management. The applications are serviced by the middleware.

1.3 Related Work

RFID provides an efficient way of automatically identifying the objects. This property of the RFID devices has enabled it to be used in many applications concerning identification. There has been some work in development of the RFID software sys-

tem. This work is manly focused in two areas, the RFID middleware and RFID applications.

1.3.1 RFID Middleware

The sudden growth of RFID in the application space has generated quite an interest in the RFID middleware development. Most of the middleware solutions which are available today are commercial in nature. A few middlewares from commercial players are Microsoft BizTalk RFID[27], Oracle Fusion[23] and Sun RFID Middleware[20]. There are also some middlewares which have been developed in the research field, of which some prominent ones are - WinRFID[31] by UCLA and Accda[13] by ETH Zurich.

Sun provides a Java based middleware platform called the Sun Java System RFID software. The Sun middleware is a part of the Java Enterprise System (JES) and supports standards-based integration with leading enterprise integration servers, including the Sun Java Enterprise Integration Server. The four components of this middleware are the RFID Event Manager, the RFID Management Console, the RFID Information Server, and a software development kit (SDK). The RFID Event Manager is a Jini-based event management system that facilitates the capture, filtering, and eventual storage of events generated by RFID readers. The RFID middleware console provides a browser based management interface, which allows configuration of various attributes and parameters of the middleware. The information server is responsible for storing and querying the EPC related data, it also manages inter Enterprise handling of the data. The SDK provides a development platform the build custom applications. The Sun middleware exposes to the application, the hardware as logical readers. These logical readers may be a collection of one or more physical readers. The application can then select one or more logical readers and apply the various processing parameters to the group.

Microsoft also provides a .NET platform based RFID end to end solution known as the Biztalk RFID Server. The Biztalk RFID middleware supports device abstrac-

tion and management to help customers manage and monitor devices in a uniform manner. Their "plug and play" architecture allows standard or non-standard devices to be supported. The middleware server supports an event processing engine that allows creation of business rules and manage the RFID events. It also enables enterprises to integrate RFID events into the business process server and provide real time visibility of the RFID data.

WinRFID is an RFID middleware from University of California Los Angeles (UCLA). Their study reports different challenges and the research approach in developing a RFID middleware to provide an efficient solution in connecting to the enterprise network. The important features discussed in WinRFID are the encapsulation of communication details, large-scale network management, intelligent data processing and routing, hardware and software interoperability, system integration and system extendability. The WinRFID middleware is supported by novel algorithms and data representation schemes capable of processing large amounts of data, rectifying errors in real-time, identifying patterns, correlating events, reorganizing and scrubbing data and recovering from faults and exceptions. It provides the support for simultaneous working of readers and tags at different frequencies, using different protocols through a layer transparent to the applications. An XML based framework in the middleware helps the filtered data from the RFID tags data streams to be formatted as per the custom plug-ins, which can be added to the middleware easily.

Accada by ETH Zurich is an RFID prototyping platform which facilitates RFID application development. The Accada platform manages readers, filters, aggregates RFID data, and helps interpreting the RFID data in the application's context. The Accada infrastructure uses EPCglobal based specifications for the reader protocols[14, 2], the application level event specifications and the EPCIS[19] capture and query interface to handle RFID data flow of across enterprises. The platform consists of three main modules – the reader, the filtering and collecting middleware and the EPC information services module. The Accada reader implementation uses

the standard edition of SUN Java Virtual Machine rather than a micro edition. This forces the reader implementation to be embedded only on the devices with significant computing resources. Also, there are a number of features which are optional and yet to be defined in the EPC Reader Protocol 1.1[18], which reflects on the reader module of Accada, making the application development a significant challenge.

1.3.2 RFID Applications

RFID is being used widely in a number of applications. Some institutions where RFID is being extensively deployed are Wal-mart[38], Speedpass[9], postal system of various countries[29], toll collection at highways[5], animal tracking[11], library management[6] etc. A number of pilot projects like construction tool tracking[17] pertaining to tracking of construction tools, mobile healthcare service system[25] used to track people inside and outside a hospital have all been implemented successfully. One of the most widely used application is the access control systems, where RFID based plastic cards are used to identify and authenticate the card-holder's entry to the facility. The RFID systems are extensively used in the warehouses and stores for the supply chain management, inventory management and movement management. This has led to huge increase in the efficiency of the warehouse operations and keeping the optimum inventory in the stores.

Pharmaceutical companies have been grappling with the high increase in the drug counterfeiting. Thus, the pharmaceutical industry reports that it loses $2 billion per year due to counterfeit drugs[24, 1]. Besides the financial losses to the industry, counterfeit drugs adversely affect people's lives by preventing patients from receiving needed medication. There has been a lot of efforts to fight this menace by using many different technologies including bar-codes, holograms, etc. RFID tags helps in detecting products that are counterfeit or fake. It also helps to identify tampered with, adulterated or substituted drugs.

RFID also finds a use in library management[6]. The various operations done in the library management includes circulation, shelf management and sorting of books.

In the current mode of operation, the books are attached with bar codes which are used to identify a book. The information stored in these barcodes generally contains the identification number. The circulation and sorting still requires a lot of human intervention. RFID presents a very good solution for library management. The books being RFID tagged, are taken out and returned many a times, thus the RFID tag is re-used again and again.

1.4 Organization of the Report

The organization of the rest of the report is as follows. In chapter 2, we present the background of RFID middleware and a few design constraints. We give a detailed architectural description of the SmartRF – the RFID middleware, in chapter 3 and talk about the implementation details of SmartRF, describe the communication architecture, initial configurations and various system requirements in chapter 4. We then describe an RFID system built using SmartRF for the postal bag tracking system in chapter 5. Finally, we conclude this thesis and provide results on the middleware evaluation in chapter 6. The appendix provides the list of APIs supported by the SmartRF middleware and few other middleware related information.

Chapter 2

RFID Middleware

An RFID middleware is the software subsystem which sits between the RFID hardware and the RFID applications. It is an interface between the software components and the hardware components of the RFID system. An RFID middleware provides certain advantages.

- It insulates applications from the RFID hardware.
- It handles the huge raw RFID tag data read by the RFID readers and processes the data before passing on the aggregated data to the applications.
- It provides an application-level interface for the uniform management of the RFID readers and querying the RFID data.

The RFID middleware provides the application with standard interfaces to access the RFID hardware. Every RFID reader has its own proprietary device driver to access its functionality. If the application is provided with the device drivers of all the connected readers, it will be a hard job to manage and interface each of the devices. In such a design the application writer will need to understand all the hardware specific internals and operations. A software layer of the RFID middleware incorporates all the device drivers of different hardwares and exposes to the application a standard set of interfaces to access any hardware. Secondly, the amount of raw tag data, which is read by the readers is huge. For example, consider an RFID system with 3

UHF readers continuously reading RFID tag data at the rate of 20 tags per second. Considering, each data item to be of 12 bytes, the amount of data read by all the three readers in one second is (3 readers x 12 bytes x 20 tags) = 720B, or, 5.76Kbps. An application if provided with this data, will find it very difficult to process it in real time. An RFID middleware can process this raw tag data and provide the application with clean and filtered data. The RFID middleware also provides a standardized way of dealing with the flood of information created by the tiny RFID tags. The application-level interface provides application-level semantics for the collection of RFID data from the RFID tags.

2.1 Middleware Components

An RFID middleware generally consists of four major components or layers (figure 2.1).

1. Reader Interface
2. Data Processor and Storage
3. Application Interface
4. Middleware Management

2.1.1 Reader Interface

The Reader interface component of the middleware is responsible for handling the interaction with the RFID hardware. It is the lowest layer of the RFID middleware. This layer maintains the device drivers for all the devices supported by the system. It manages all the hardware related parameters like the host-side communication interface of the reader, the air interface of the reader, etc.

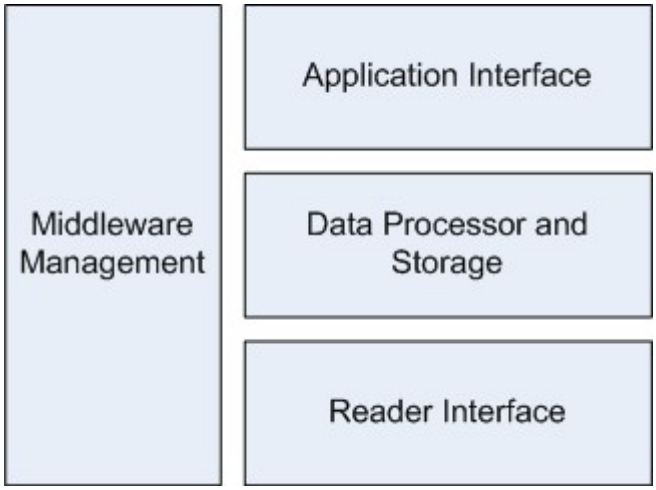


Figure 2.1: RFID Middleware Components

2.1.2 Data Processor and Storage

The Data processor is responsible for handling and processing the raw data coming from the readers. This component is also responsible for storing the raw tag data, so that it can be processed. Some of the important processing logic carried out here are filtering and grouping of the RFID data. This component also manages the data level events associated with the application. As an example, when the applications requests for all data capture between a time interval, the processing of the time stamp is carried out by this component and then the data is given to the application.

2.1.3 Application Interface

The Application interface component manages the middleware's interface with the application. It provides the application with an API to communicate and configure the RFID middleware. It accepts requests from applications and translates them down to the underlying components of the middleware. This component is responsible for the integration of the enterprise applications with the RFID middleware.

Most of the commercial players which provide the middleware, not only provide these core components but also certain enterprise level solutions. These solutions help building complete RFID systems.

2.1.4 Middleware Management

The middleware management component helps in managing the configuration of RFID middleware. It provides information on all the processes running in the middleware. The middleware management provides the administrator the following capabilities.

1. Add, remove and modify the RFID readers connected to the system.
2. Modify various parameters like filters which were configured by the applications.
3. Enable and Disable various features supported by the middleware.

RFID middlewares typically provide certain abstraction to the application layer. Usually, readers are abstracted as logical reader which may be a collection of several readers or a part of the reader. Logical reader is a name associated to the group of readers. This is a grouping mechanism provided by most middlewares where there is a need to have a set of readers capturing observations from a particular area. For example, in a warehouse with 10 loading docks, each of which had a couple of readers, can be grouped under one or more logical readers. The application can then query a small number of logical readers for incoming product information rather than having to aggregate events from each of the individual readers.

The applications are usually provided with a standardized interaction model with the middleware. These interaction models define the communication between the middleware and the application clients. A client can either request services on demand (synchronous mode) or register for information to be sent to it when certain conditions are met (asynchronous mode). In the synchronous mode of operation, the client requests for a particular service from the middleware and suspends execution till it receives a response. In the asynchronous mode of operation the client subscribes to events at the middleware. The middleware can then asynchronously delivers data back to the clients.

The RFID tag data is usually needed to be filtered to remove unwanted information. Filtering provides the capabilities to tune into specific patterns in the data. Sometimes, it might be required to report only certain type and value of the tag data to the client. RFID middlewares usually provide some kind of data filtering mechanism. Here, the client provides a set of pattern in a defined format to the middleware. The middleware then allows only that data which matches the pattern, to be reported to the client. For example if an application needs to see all tag data which start with a specific pattern such as 19AD, the associated filter for this can be provided by the client as - [19AD****].

2.2 Middleware Design Issues

The RFID middleware must have a robust and flexible design. Some of the issues to be considered during the design of an RFID middleware are the following.

1. Real time handling of incoming data from the readers – Huge amounts of RFID tag data flows into the middleware from the various readers connected to it. The middleware should be able to process this real-time data without read misses.

2. Multiple Hardware Support – The middleware must support multiple kinds of tags and readers by providing a common interface to access them. Different RFID hardware offer different features, the middleware must be in a position to access all the features and capabilities of the hardware.

3. Synchronization and Scheduling in the middleware – The middleware is typically realized using multiple processes for handling readers, applications, data processing elements and buffers. There should be intelligent scheduling and synchronizing among all these processes. These factors define the latency and efficiency of the middleware.

4. Servicing Multiple Applications – The middleware design must be capable of servicing multiple applications simultaneously. Different applications have different requirements, the middleware must cater to all the requirements of the applications with minimal latency.

5. Device Neutral Interface to the applications - The middleware should provide the application a device independent view of the hardware. The application writer should be able to develop applications using only the generic set of interfaces provided by the middleware. The applications development should be totally independent of the type of hardware connected to the system.

6. Scalability – The middleware design must allow new hardware to be easily integrated in the RFID system. This calls for a modular design, where modules can be added or removed based on various configurations. The design must also allow easy integration of newer data processing features into the middleware.

Chapter 3

SmartRF – The RFID Middleware

In this thesis, we developed an RFID middleware called SmartRF keeping in mind the design issues discussed in the previous chapter. Our RFID system is organized as a three tier architecture (figure 3.1) with applications, middleware (SmartRF) and the RFID hardware. The RFID hardware represents the RFID readers and the tags. The middleware (SmartRF) is a software component which provides the RFID applications with a device independent interface to access the RFID hardware. The RFID applications are independent softwares which use the services of SmartRF for various consumer requirements. These applications are developed using a generic application framework. This framework uses a device independent visualization of the RFID hardware provided by SmartRF to build applications.

The RFID readers generally have multiple antennas connected to it. These antennas may be placed in groups at different locations for tracking purposes. A group of antennas at one tracking point may belong to single or multiple readers. There may also be readers whose antennas are part of various groups allowing a single reader to be a part of multiple tracking points. SmartRF renders this functionality by introducing a notion of channels which allows us to combine multiple reader-antenna pairs. The channel is a virtualization of a tracking point and is used to associate multiple reader-antenna pairs to the tracking point.

SmartRF has a novel design which provides the application a device neutral,

protocol and platform independent interface. It incorporates three subsystems – Hardware abstraction layer (HAL), Event and data management layer (EDML) and Application abstraction layer (AAL).

3.1 Hardware Abstraction Layer (HAL)

The HAL is the lowest layer of SmartRF and is responsible for interaction with the RFID hardware. It provides access to the devices and tags in a manner independent of their various characteristics through tag and reader abstraction layers.

SmartRF provides the view of tag data as a stream of bytes. This level of tag abstraction provides independence from various tag characteristics like protocols (ISO 14443[2], EPC Gen2[14], ISO 15693[3], etc.), air interfaces[16] (HF, UHF) and memory sizes[12]. The reader abstraction provides a common interface to access the RFID hardware devices with different characteristics such as protocols (ISO 14443, EPC Gen2, ISO 15693), air interface (UHF, HF) and host-side communication interface (RS232, USB, Ethernet). The reader abstraction exposes simple functions like open, close, read, write, etc. to accomplish complex operations of the readers. The reader and tag abstractions in SmartRF make it extendable to support various tags and readers.

The Device Management Module in the HAL is responsible for dynamic loading and unloading of the driver libraries depending upon the usage of the hardware devices. This allows the system to be light weight as only the required libraries are loaded. This layer configures the devices for various operations as specified by the upper layers. It is also responsible for monitoring and reporting the device status.

Some of functions provided by the HAL to access the RFID hardware are the following.

- OpenDevice – Open device function is responsible for opening a connection with the device. The connection parameters are provided as an argument to this function. On a successful connection with the reader, a handle is returned

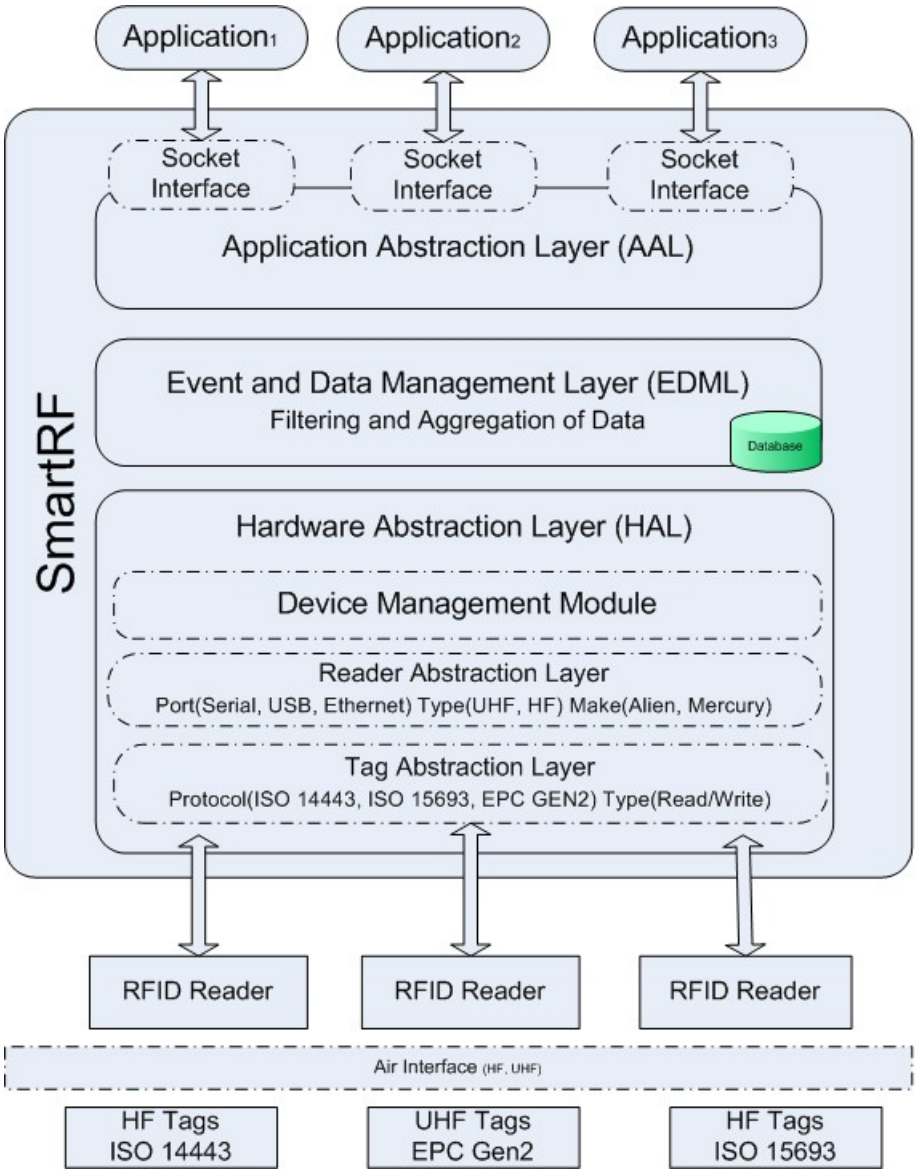


Figure 3.1: SmartRF System Design

by this function. This handle is then used as a reference to access the device in subsequent calls.

- ReadDevice– Read device function reads the data from the reader's internal buffer. The read parameters like the protocol to be read by the reader, the size of data to be read, are specified as arguments to this function. The function returns successfully with data if valid data is present in the reader buffer or with an error code.

- WriteDevice – WriteDevice function writes data to the tag. The arguments specified to this function are the unique ID partially or wholly, which identifies the tag to be written, and the data to be written to this tag. The function returns successfully if data is written to the tag or returns an error code (for example when the tag is not identified uniquely).

3.2 Event and Data Management Layer (EDML)

The EDML manages various reader-level operations such as reading tags and handles reader notifications such as device-disconnected, tag-write-failures, etc. The EDML layer acts as a conduit between the hardware abstraction layer (HAL) and the application abstraction layer (AAL). It accepts commands from AAL, processes them and accordingly issues commands to the HAL. Similarly, the responses are carried from the HAL, processed and passed on to the AAL by this layer.

This layer acts as a temporary storage for incoming data from the reader. It processes the data by grouping, filtering and duplicate removal as described below.

Grouping: Grouping involves accumulation of the data from various readers. Initially the readers read data into their own private buffers. In grouping, the data form these reader buffers is accumulated to a single location. This accumulation is based on the hardware configuration done by the applications. This grouped data is then used for further processing by SmartRF such as sorting them channel wise.

Filtering: There are cases where applications needs only specific tag-IDs to be

reported. Filtering provides this functionality by selective reporting of the tag data. In SmartRF abstraction tags are viewed as a stream of bytes. Therefore, we choose the filter values as provided by the application, in the form of consecutive bytes. This offers flexibility in handling multiple tag-data formats.

Duplicate Removal: Tags in the vicinity of the readers are read continuously. This results in large amount of repeated data from the readers. Duplicate removal is a feature provided by SmartRF, to prevent reporting of duplicate data. It is characterized by a time value specified by the application. Same tag data read by the reader within the specified time duration is only reported once.

3.3 Application Abstraction Layer (AAL)

The Application abstraction layer (AAL) provides various applications with an interface to the RFID hardware. The interface is designed as an API through which the applications use services of the SmartRF. All the application level operations such as read, write, etc. are interpreted and translated to the lower layers of the SmartRF by the AAL.

The AAL provides an abstraction of the hardware as data streams and channels (figure 3.2).

3.3.1 Data Streams

An RFID reader typically connects to multiple antennas. A tag might be seen by one or more antennas depending upon the air interface and tag illumination. SmartRF therefore defines the concept of data stream as a reader-antenna pair. The number of data streams for a reader is equal to the number of antennas connected to it. Thus, every data stream acts as an independent source of data received by the middleware from the reader. SmartRF exposes to the applications all the available data streams. For example, in a system with two readers R_1 and R_2 with antennas A_{10}, A_{11}, A_{12}, A_{13} and A_{20}, A_{21}, A_{22} connected to them respectively, SmartRF will provide the

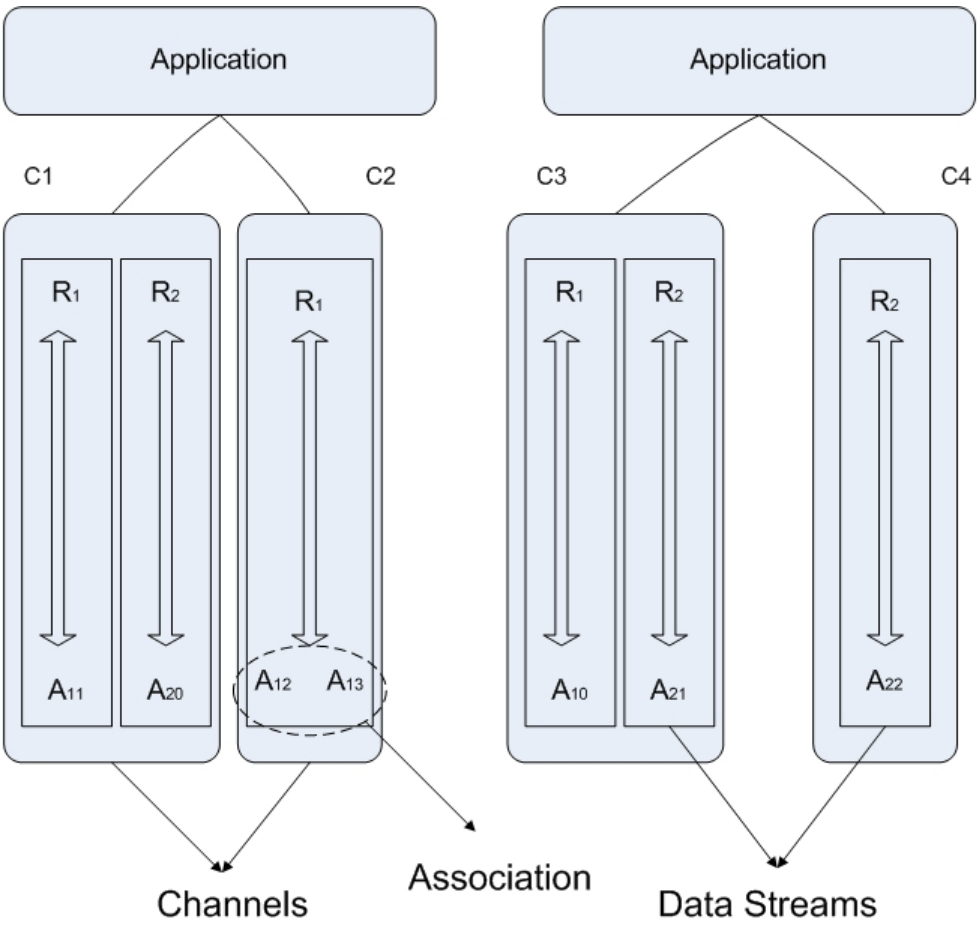


Figure 3.2: Data Streams, Channels and Association

following data streams (R_1, A_{10}), (R_1, A_{11}), (R_1, A_{12}), (R_1, A_{13}), (R_2, A_{20}), (R_2, A_{21}) and (R_2, A_{22}).

There are some readers which use different antennas to transmit and receive RF signals. This puts a limitation on the usage of a single antenna as an independent data source. This limitation is incorporated in SmartRF by introducing the concept of 'Association'. This concept forces the dependent antennas (transmitter and receiver) to be grouped together forming a single data stream. Such a grouping may be based on illuminator and communicator or any other related pairing.

3.3.2 Channels

The channels are representation of an RFID gate that incorporates one or more data streams (i.e reader-antenna pairs). A channel is a logical grouping of different data streams. Applications define channels by grouping data streams as per their requirements. For example, if a door has two antennas placed orthogonally, the channel will be defined to incorporate both of these data streams. Multiple data streams in a channel may belong to same or different readers. The data streams in the channels are used in a mutually exclusive manner. Once a data stream is selected as a part of a channel, it cannot be used further by other channels, until the channel is destroyed. An application can create any number of channels as long as there are data streams available.

During the creation of a channel, the associated antennas are automatically pulled in as a part of the channel, whenever an antenna is selected that is a part of an association.

An example of channels and data streams is given in figure 3.2. In the figure, R_1 and R_2 are the readers connected with A_{10}, A_{11}, A_{12}, A_{13} and A_{20}, A_{21}, A_{22} antennas respectively. In this example, four channels $C1$, $C2$, $C3$ and $C4$ are created here. Channel $C1$ and $C3$ have two data streams each, whereas channel $C2$ and $C4$ have one data stream each. The concept of association is shown by channel $C2$. Here,

antennas A_{12} and A_{13} are part of an association. Thus when A_{12} or A_{13} is selected as part of a channel, the other associated antenna is automatically selected as a part of the channel.

In the APIs of the AAL, channels are indicated by a channel handle. The channel information is maintained by the SmartRF and only the handle is returned to the application at the time of channel creation. Subsequent function calls for read/write/delete channel use the channel handle. The channels are independent of the type and nature of the applications connected to it. This makes SmartRF scalable to support a variety of applications.

Some of the functions which are available to an application as provided by the AAL are the following.

- CreateChannel – CreateChannel function specifies a list of reader-antenna pairs to be grouped together in a single channel. A handle identifying the channel is returned to the application for further communication. This call translates down to EDML, where SmartRF keeps a mapping of the channel and the corresponding reader-antenna pairs. The EDML is responsible for aggregating tag-data from the reader-antenna pairs belonging to a single channel.

- Read_data – Read_data function reads the data from the channel. SmartRF replies back with the tag data as buffered by the EDML for the specified channel. This function call supports blocking as well as non-blocking read operations. In the case of a blocking call, SmartRF replies to the application with tag data for that channel. If the tag data is not available the function waits for that to become available. In the non-blocking call, the Read_data replies to the application immediately with data or a status indicating that data is not available.

- Destroy_channel – Destroy_channel function deletes the channel specified by the application. All the reader-antenna pairs associated with this channel are subsequently made available for grouping into another channel.

- Configure_channel – Configure channel function configures various channel specific parameters like filter masks, time window for duplicate removal and data aggregation specification.

- Write_data – Write_data function writes data to the specified channel. SmartRF translates this call to a tag-write command. This is done by identifying the tag to be written, and then writing the required data to the tag. In the case of multiple tags being identified the Write_data proceeds without writing data to the tags and returns a suitable status.

Chapter 4

SmartRF Implementation

SmartRF – The RFID middleware is an open-source and platform-independent middleware. The computing architecture used by SmartRF is a Client -Server based distributed system. It internally uses a multi-threaded architecture to handle different middleware processes.

4.1 Implementation Model

SmartRF uses the Client Server Computing model for external communication and Multi-threaded model for internal implementation.

4.1.1 Client-Server Model

The Client-Server computing model of SmartRF (figure 4.1) consists of two independent softwares, the middleware server and the application clients. An application client initiates a communication session while the server waits for a requests from the client over a socket interface. This computing model allows flexibility as the client can be designed independently and can even reside on a remote host. In such a case, communication between the middleware and the application takes place over the network. In SmartRF, the network based communication also takes place through sockets. The middleware creates a socket and listens for incoming client connections on a pre-defined TCP port (11002 in our implementation). The client

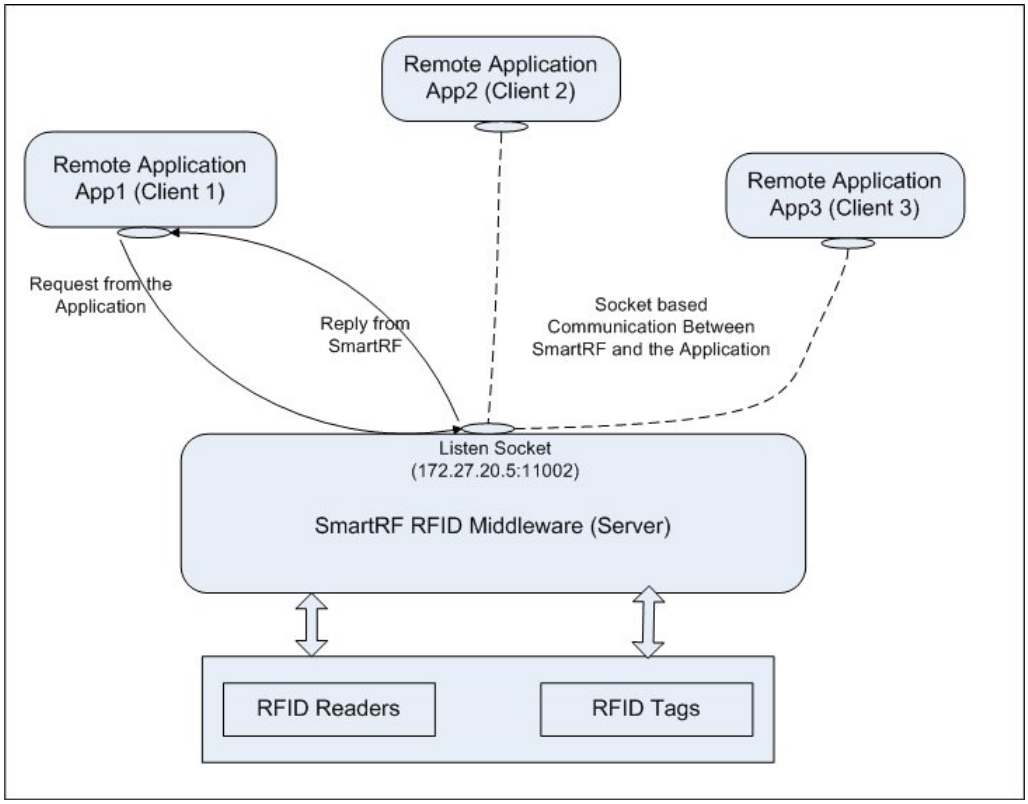


Figure 4.1: Client-Server Model of SmartRF

applications can then initiate a connection with SmartRF by connecting on this port. The application and the middleware communicate by sending data packets containing specific commands and responses in a pre-defined format. On receiving a packet from the application, SmartRF parses it, and decodes the incoming command and other packet parameters. It then executes the required functionality and replies back to the application.

4.1.2 Multi-threaded Architecture

SmartRF internally uses a multithreaded architecture (figure 4.2) for handling various actions. After the middleware is set for operation, a thread is created that listens forever for incoming connections from the applications. For every application that connects to this thread, a new thread is spawned by the middleware. The new thread continues to serve the requests from the application while the older thread goes back to listening for new connections. For application requests pertaining to the global

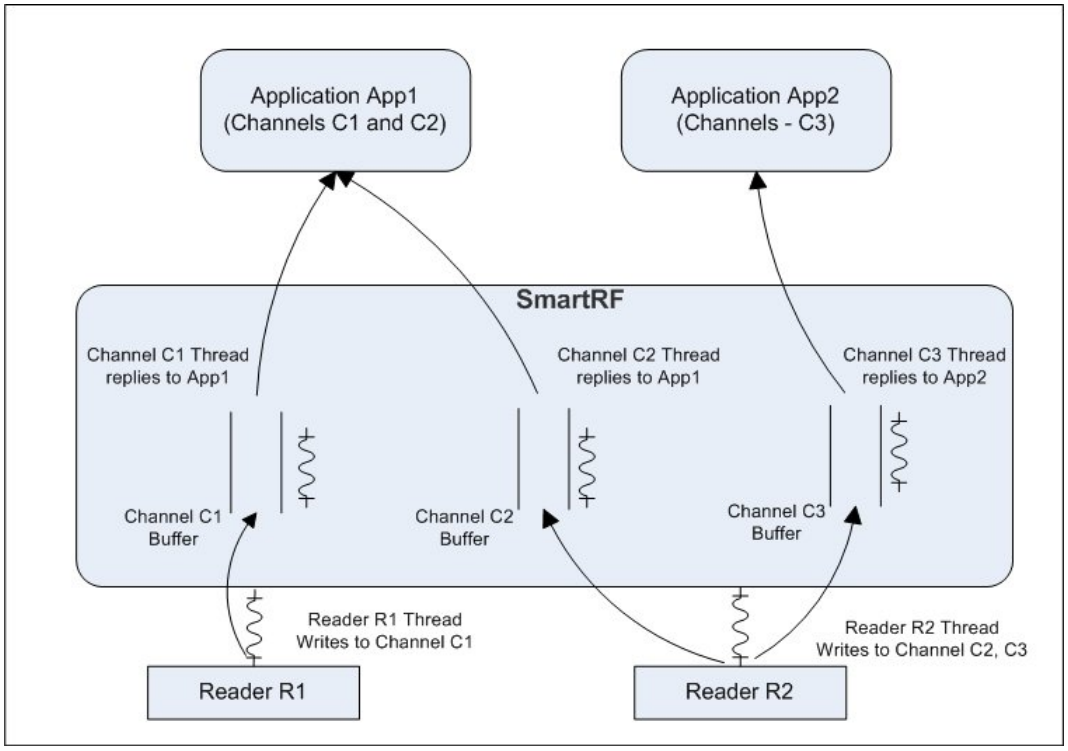


Figure 4.2: Multi-threaded Architecture of SmartRF

middleware information like the request to get the list of readers connected, or to get the list of channels created, etc., the newly created thread gives the response and terminates. For channel specific functions like Create_Channel, which involves setting up a separate data exchange interface between the the application and the SmartRF, the newly created thread services this channel, until the channel has been destroyed. Thus, for every channel created by the applications, a dedicated channel servicing thread is created by the SmartRF. Every channel is assigned a buffer in SmartRF which stores channel specific data. On a read or on a write request from an application, the channel specific thread handles the corresponding request.

In addition to the channel specific threads, a dedicated thread is created internally for each reader connected to the SmartRF. This thread, known as the reader thread, is responsible for collecting data from the reader buffer. The reader thread also redirects the data to the channel buffer based on the channel-data stream configuration.

Thus, the SmartRF has the following types of threads during its execution.

- Application Listening Thread - This thread listens for incoming connections from the application.
- Channel Servicing Thread - This thread services one channel.
- Reader Servicing Thread - This thread handles and communicates with one reader.

The synchronization constructs used by SmartRF are Semaphore and Mutex. Synchronization is needed between the following threads.

1. Between the Reader Servicing Threads - Every reader has its own thread in SmartRF. All readers which are part of a particular channel will compete to access the channel buffer. These threads will be responsible for writing data into the channel buffer.
2. Between Channel Servicing Thread and Reader Servicing Thread - The Channel Servicing Thread reads data from the channel buffer and passes on that to the application. Thus, there is a need of synchronization between the reader servicing threads that write to the channel buffer and the channel servicing thread that reads from the channel buffer.

This problem of synchronization is similar to that of classic multiple writers, single reader problem.

4.2 HAL Implementation

SmartRF provides the RFID applications an interface to the RFID hardware. This access to the hardware by the SmartRF is carried out by the Hardware abstraction layer (HAL). Generally, the RFID readers are accessed by a set of APIs provided by the reader manufacturer, which are specific to each reader. These APIs are provided as DLLs or Shared Objects, as a part of the RFID reader package. For any

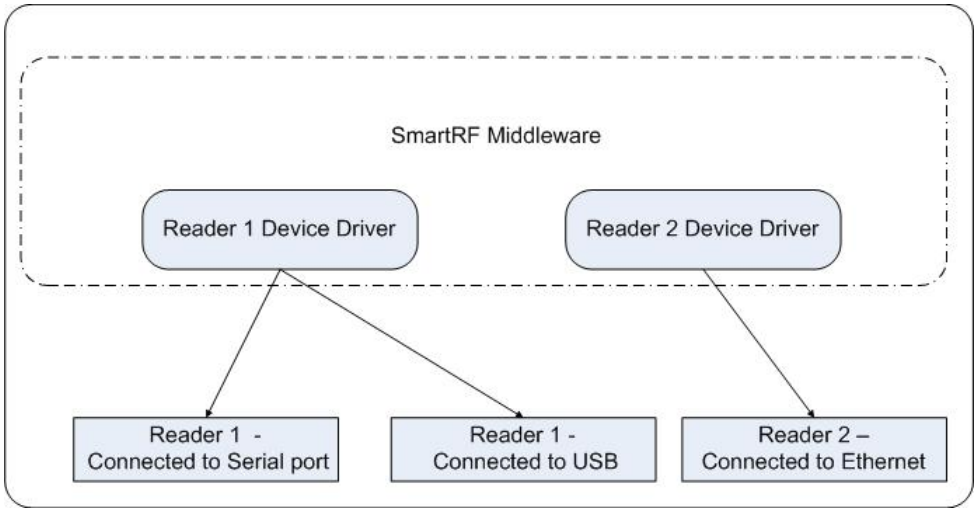


Figure 4.3: HAL Device Driver

reader to be accessible through SmartRF, it must implement the set of APIs defined by SmartRF. Thus, in our implementation we use wrappers for the manufacturer-provided reader APIs to make the readers accessible through SmartRF. These wrappers internally call the reader specific API to implement the desired functionality.

Readers which are accessible through SmartRF have their drivers in the form of DLLs or shared objects as part of the HAL. If there are multiple similar kind of readers connected to the middleware, then all accesses to these readers takes place through a single instance of the reader driver (figure 4.3). For example, if one reader (Reader 1) is connected to the system on the serial port interface and the other similar reader is connected via the USB interface, both these devices will be accessed using the same device driver. The driver is internally responsible for maintaining a mapping between the connection type of the reader (Serial Port, USB, Ethernet) and the API to be called internally. During the initial connection setup, the driver returns a reader handle, which identifies the type of connection. SmartRF, then uses this handle for all further communication with this reader.

The HAL carries out this functionality by loading the manufacturer specific reader libraries on demand. The information regarding these drivers (such as the file name, its location on the file system) is provided in the SmartRF configuration file. The SmartRF scans this configuration file in the beginning at the start up time.

It then identifies the DLL and dynamically loads them for each device supported by the SmartRF.

4.2.1 HAL Driver Functions

The HAL wrappers for the manufacturer-specific reader drivers interface to the SmartRF middleware. For this, a wrapper must provide certain functions which are called by the HAL layer. The following are the functions that must be implemented by the wrappers for device drivers for the RFID readers. These functions constitute the HAL API.

4.2.1.1 Open Reader

The Device_Open_Reader function is responsible for initiating a connection with the reader. If the reader can be successfully connected, a handle identifying the connection is returned otherwise the function returns a NULL value. All further operations for this connection are identified by the reader handle. This function must take as an input the connection type of the reader. The connection type in SmartRF is identified by Port_Location:Port_type. For example the valid connection types are COM:COM1, IP:172.27.20.5. More details on this are given in the Appendix.

Prototype:

void * Device_Open_Reader(**char** *Connection_Identifier)

4.2.1.2 Close Reader

The Device_Close_Reader function call is used by SmartRF to close the connection with the reader. This function must take the reader handle as an input argument. On the successful execution of this function call, all reader operations are stopped and the reader handle becomes invalid. The function returns with a success code upon successful execution or with an error code.

Prototype:

int Device_Close_Reader(**void*** Reader_Handle)

4.2.1.3 Configure Reader

The Device_Configure_Reader function is used by SmartRF to set the reader specific parameters like the power level of operation, antenna operation specifications, etc. The input arguments to this function must be the reader handle and a configuration structure which specifies the list of the parameters and their values to be set by the driver. The reader specific list of parameters to be set must be initially specified in the SmartRF configuration file. SmartRF parses this configuration file in the beginning and configures the device using this API. The function returns with a success code upon successful execution or with an error code.

Prototype:

int Device_Configure_Reader(**void *** Reader_Handle, **struct Config_Params** Config)

4.2.1.4 Start Read Operation

The Device_Start_Read function is used by SmartRF to start the read operation at the reader. The wrapper must implement this function to be a continuous read operation, until explicitly stopped by the HAL. The function must accept as input arguments the reader handle which identifies the reader connection and a structure specifying the read parameters – the size of data to be read by the reader and the protocol used by the reader for tag identification. This function only initiates the read operation. The driver or the wrapper must internally store the data read by the reader and return it later using the Get Data operation. The function must return a success code or an error code depending upon whether the operation was successful or not.

Prototype:

int Device_Start_Read (**void *** Reader_Handle, **struct Read_Param** Params)

4.2.1.5 Get Data

The function Device_Get_Data is used by SmartRF to read data from the driver or the wrapper buffer. The input arguments to this function provided by SmartRF are the reader handle and a pointer in memory which is used to return the data upon successful execution of the function. The function call is expected to return the size of data stored in the buffer and this value may even be zero.

Prototype

int Device_Get_Data(**void *** Reader_Handle, **struct Get_Read_Info** *Params)

4.2.1.6 Stop Reader Read

The Device_Stop_Read function is used by SmartRF to stop ongoing read operation of the reader. The reader handle is provided by SmartRF as an argument. The driver returns a success code on the successful execution or an error code specifying the error.

Prototype

int Device_Stop_Read(**void *** Reader_Handle)

4.2.1.7 Number of tags available

The Device_Num_Tags function implemented by the driver/wrapper must return the number of tags available in the reader buffer which are not yet read by SmartRF. This helps SmartRF to determine the number of read operations needed, to empty out the reader buffer. The input argument to this function is the reader handle. An error code specifying the error is returned by the driver if the function fails.

Prototype

int Device_Num_Tags(**void *** Reader_Handle)

4.2.1.8 Write Data

The Device_Write function is used by SmartRF to write data to the specified tag. The input arguments provided to this function are the reader handle and a structure

that must specify the partial or full tag ID to uniquely select the tag to be written to, the data to be written to this tag and the protocol to be used for the data write operation. A code must be returned to indicate success on the successful execution of this function or an error as the case may be.

Prototype:

int Device_Write (**void *** Reader_Handle, **struct Write_Params ***Params)

4.2.1.9 Select Antenna

The Device_Select_Antenna function is used by SmartRF to power up the antennas of the reader. The input arguments to this function provided by the SmartRF are the reader handle and a list of the antenna IDs to be powered on.

Prototype

int Device_Select_Antenna(**void *** Reader_Handle, **int** Selected_Antennas[], **int** Num_Antennas)

4.2.1.10 Set Association

The Device_Set_Association function is used by SmartRF to set associations between antennas of a reader. Some readers are unable to use the same antenna for transmit and receive operations. In such cases, the readers use a set of associated antennae to perform the RFID operation. This association may be fixed by the hardware or programmable at hardware but fixed by the installation. The illuminator antenna transmits radio waves and the receiver antenna reads the data from the tag. These two (illuminator and the receiver) antennae then form an association. This function is implemented by the driver to check if the specified association is possible. The input arguments are the reader handle and a set of transmitter and receiver antenna which are to be associated. If the association has been successfully set, a success code is returned or otherwise an error code is returned.

Prototype

int Device_Set_Association(**void *** Reader_Handle, **struct Associations** As-

sociate)

4.2.1.11 Check Reader Connectivity

The Device_Check _Connected function is used by SmartRF to check if the reader device is connected to the system. The input argument to the function must be the reader handle. A success code is returned if the reader is connected or an error code is returned to indicate an error.

Prototype

int Device_Check _Connected(**void *** Reader_Handle)

4.3 EDML Implementation

The EDML is the data processing layer of SmartRF. The main functionality of this layer is to maintain an internal database of a per-channel buffer and to process the RFID tag data in a manner configured by the application. The data into the channel buffers is inserted by the reader thread whenever it reads the value of a tag. The EDML then processes the data by the process of duplicate removal, data replacement policy, grouping and filtering.

Duplicate removal functionality allows the application to configure the time window during which same tag data read multiple times is not reported by the middleware to the application. The duplicate read time window is specified in seconds. For duplicate removal processing, there is a need to search for tag data read by the reader for existence in the middleware buffer and ignore those entries which fall within the duplicate time window. To reduce the search time, SmartRF uses a hash table. The reader thread of SmartRF, on reading data from the driver/wrapper buffer computes its hash. If the corresponding entry in the hash table is marked valid, it compares the previously stored data with the tag data. If this is indeed a duplicate, the tag data is discarded. Otherwise the tag data is inserted into the hash table when the corresponding hash table entry was invalid or did not match with the tag data. In the case of a duplicate entry, the new data is accepted only if the time gap between

the existing and the new data item is more than the duplicate time window. Thus, this allows us to search and insert data into the channel buffer in O(1) time. Any request for the data from the application is then carried out by the channel thread, by serially accessing the channel buffer and invalidating the hash table entry for the outgoing data.

The data replacement policy allows the application to specify the action to be taken by SmartRF in case of a channel buffer overflow. In SmartRF, the readers continuously insert tag data into a finite sized channel buffer. Usually, the rate at which the tag data is inserted in the channel buffer by the readers is much slower than the read rate of the applications. However when the application is slow or when the application is not executing, the channel buffer may get full. This leads to a buffer overflow problem, when there is no free buffer available for the new data read by the reader. To handle this problem, SmartRF provides the application with an ability to specify a data replacement policy to be used at the channel buffer. The following are the two policies.

1. Replace the oldest – In this policy, the oldest buffered data gets written with the new data when the channel buffer is full. Thus, the channel buffer always contains latest data read by the readers.

2. Do not replace – In this policy, the readers stop inserting new data into the channel buffer once it is filled. New data is accepted only when the old data is read by the application and a free buffer becomes available.

The filtering feature provided by SmartRF allows the application to request for selective reporting of tag data. The filtering specification can be provided by the application on a per-channel basis. This specification consists of a filter value and a mask value specified as sequence of bytes. During the filtering, SmartRF performs a bit-wise and operation between the RFID tag data and the provided mask value. It then compares the masked value with the filter value. Only the matched values of the tags are forwarded to the application. This filtering process is carried out after

the filter specifications are set by the application. In our implementation, the filter is disabled initially and thus all the data read by reader is reported to the application.

As an example consider the scenario where the application needs to specify a filter such that SmartRF returns to the application the tag data which have a value of hexadecimal 55 in their 3rd and 4th byte locations.

The filter mask for this purpose shall be 0xFFFF0000

The filter value to be matched shall be 0x55550000

Now consider that at some point in time the following tag data are present in a channel buffer.

1111 2222 3333 4444 5555 9911

1111 2222 3333 4444 2222 9911

1111 2222 3333 4444 8888 9911

1111 2222 3333 4444 5555 8888

The tags which will be selected by SmartRF after applying the filter are the following.

1111 2222 3333 4444 5555 9911

1111 2222 3333 4444 5555 8888

4.4 AAL Implementation

SmartRF is implemented as a daemon which interfaces multiple applications to multiple readers. An RFID application which uses the services of SmartRF through AAL typically performs the following operations.

- Get the list of readers connected to SmartRF
- Create / delete the channels.
- Read / write tags through the channels.
- Modify channel configuration parameters such as filtering specifications, time window for duplicate removal etc.

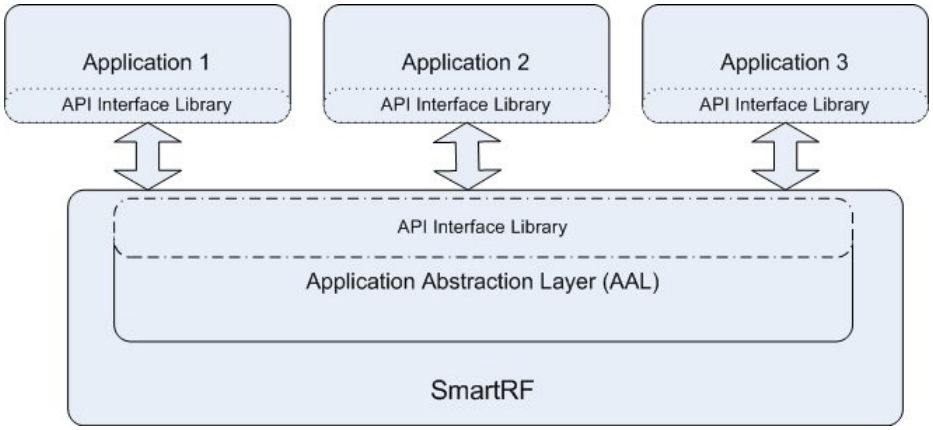


Figure 4.4: SmartRF – Application Interaction

SmartRF supports all these operations by providing the application with a set of APIs. The applications and SmartRF are separate processes which may even run on different systems communicating through a socket interface. SmartRF provides the application writer with a library which interprets the API calls and passes them to the SmartRF middleware server (figure 4.4). The general flow of an RFID application when it needs the services of SmartRF is as follows.

1. Connect to the SmartRF daemon.
2. Use services through the APIs provided by SmartRF to configure/access devices.
3. Close the connection with SmartRF (optional).

4.4.1 AAL API Specification

The following is the API which is provided to the applications by SmartRF to use its services.

4.4.1.1 Connect to Middleware

The connect_app function is used by the application to establish a connection with SmartRF. The input argument to this function are the IP address with the port

(specified as a single argument – 'IP:port') on which SmartRF listens for incoming connections and a pointer to the connection descriptor variable that holds the information about the type of connection. The function returns a success code if the connection is successful or an error code when it is unsuccessful.

Prototype:

int connect_app(**string** ip_address_port, **Connection_descriptor** *mwConnection)

4.4.1.2 Get Reader List

The Get_Reader_List function is used by the application to get the list of readers connected to the SmartRF. The input arguments to this function are the connection descriptor identifying the connection and a pointer to memory where the list of readers is returned that are connected to the SmartRF. The function call returns the total number of readers connected to SmartRF. If the function fails, an error code is returned.

Prototype

int Get_Reader_List(**Connection_descriptor** mwConnection, Reader_Info **Reader_list)

4.4.1.3 Create Channel

The Create_Channel function is used by the application to create a channel. The input arguments to this function call are the connection descriptor identifying the connection, a list of data streams to be a part of the channel and the operation type (read / write) of the channel. The function returns a channel ID if the channel creation is successful or an error code specifying the error. The application then uses this channel ID for any subsequent communication related to this channel.

Prototype

int Create_Channel(**Connection_descriptor** mwConnection, **string** Channel_name, **Reader_antennas*** Data_streams, **int** Operation_type)

Parameter ID	Type	Description
PARAM_NUM_BYTES	integer	Maximum number of bytes which can be read by the Read_data function call
PARAM_DUPLICATE_TIME	integer	Duplicate window time in seconds
PARAM_MASK_VALUE	string	Mask value for the channel filter
PARAM_FILTER_VALUE	string	Filter value for the channel filter
PARAM_DATA_REPLACE	integer	Data replacement policy used by the middleware for the channel buffer overflow. The policies are described in section 4.3.

Table 4.1: Channel Parameters

4.4.1.4 Change Channel Configuration

The channel_config function is used by the application to change the existing channel configuration. Several parameters are available for the channel, each of which may be changed through this function call. The table 4.1 lists the parameters for the channel and their meaning.

The input argument to this function are the channel handle identifying the channel and a structure specifying the list of parameters to be changed. The function returns a success code or an error code depending upon whether the requested set of parameters are changed successfully or not.

Prototype

int channel_config(**int** Channel_Handle, **Channel_Param** Params)

4.4.1.5 Get Channel Configuration

The Read_Config function is used by the application to get the values of parameters for a channel from SmartRF. The table 4.1 gives the list of parameters whose values are returned by SmartRF through this function call. The input arguments to this function are the channel handle and a pointer to the memory where the list of channel parameters is returned. The function returns a success code if the call is successful or an error code when it is unsuccessful.

Prototype

int Read_Config(**int** channel_handle,**Channel_Info** **config_info)

4.4.1.6 Destroy Channel

The Destroy_Channel function is used by the application to delete the specified channel from SmartRF. After the channel is destroyed, its corresponding data streams are available to become part of another channel. The input argument to this function is the channel handle which identifies the channel. This function returns a success code if the specified channel is deleted or an error code which specifies the error.

Prototype

int Destroy_Channel(**int** Channel_handle)

4.4.1.7 Read Data

The Read_data function is used by the application to read data from SmartRF. The input arguments to this function are the channel handle that identifies the channel, a pointer to memory where the tag data read is returned, another pointer to the memory where the time stamp of the data read is returned. In addition, this function also takes another argument as the type of the read operation – blocking or non-blocking. In blocking read, SmartRF returns to the application only when the valid data is available. Whereas, in non-blocking read, SmartRF returns immediately either with data if it is available or without data if it is not available. This function returns the size of data (which may be 0 if no data is available) or an error code in case of a read failure.

Prototype

int Read_data(**int** Channel_handle, unsigned char *buffer, **ssize_t** *time_data, **int** read_type)

4.4.1.8 Write data

The Write_data function is used by the application to write data to a selected tag. The input arguments to this function are the channel handle that identifies the channel, the tag ID value which partially or fully identifies the unique tag to be written to, and, the data to be written to the tag. SmartRF first checks for the

tag IDs present in the range of the selected channel. If a unique tag matches the specified tag ID value, then the data is written to that tag. If more than one or no tags match the given tag ID, an error code is returned. The function returns a success code if the data is successfully written to the tag.

Prototype

int Write_data(**int** Channel_handle, **unsigned char** *partial_tag_id, **int** tag_id_length, **unsigned char** *data, **int** data_length)

4.4.1.9 Exit Application

The Exit_Application function is used by the application to close its connection with SmartRF. The input argument for this function is the connection descriptor that identifies the application. SmartRF upon receiving this call deletes channels created by this application. The use of this function is optional. The applications when terminate without calling this function also result in the deletion of channels.

Prototype:

void Exit_Application(**Connection_descriptor** mwConnection)

4.5 Initialization of SmartRF

The initial configuration of SmartRF is defined in an XML file, a sample of which is given in figure 4.5. The configuration files gives information of all the readers which are be connected to SmartRF. This file has a hierarchical structure.

The 'config_info' tag is the root level tag and encloses all information described in the configuration file. The 'config_info' tag contain one or more 'reader' tags. Each of these 'reader' tags contain the reader specific information for one reader. Thus, the number of readers in the system are equal to the number of 'reader' tags in the configuration file. The following are the list of tags defined inside the 'reader' tag.

1. manufacturer_name – This tag specifies the manufacturer name of the reader. In figure 4.5 'ABCDEF' is the manufacturer name of the reader.

Parameter ID	Possible Values	Description
IP_ADDRESS	Valid IP address of the form x.x.x.x. Example – <name>IP_ADDRESS</name> <value>172.24.24.19</value>	IP address of the reader
BAUD_RATE	Valid serial port baud rate. Example – <name>BAUD_RATE</name> <value>115200</value>	Serial port baud rate supported by the reader

Table 4.2: Reader port parameters

2. username – This tag specifies the user name of the reader. Usually, this name is the location at which the reader is placed. The applications identify the readers through their name. In figure 4.5, 'CSE Lab' is the user name of the reader.

3. port – This tag specifies the port information on which the reader is connected to the host system. The 'port' tag contain one or more 'param' tags, each of which describes one port parameter. The tags 'name' and 'value' provide the parameter information within the 'param' tag. The table 4.2 provides information regarding the various parameters and their possible values. In figure 4.5, the name of the parameter is 'IP_ADDRESS' and its value is '172.27.22.36'.

4. antennas – This tag provides information regarding all the antennae connected to the reader. A reader might have multiple antennae. Thus, there are multiple 'antenna' tags each of which provide information regarding one antenna connected to the reader. The antenna information such as the antenna number of the antenna as identified by the reader (antenna-id), the user defined name of the antenna (username) and the port name of the reader on which the antenna is connected (reader-port) are specified within the 'id', 'username' and 'reader_port' tags respectively. In figure 4.5, there is one antenna connected to the reader whose id, user name and reader port are '1', 'top' and 'port1' respectively.

5. association – This tag provides information regarding the antennae association. The 'association' tag contains the 'operation_type' tag, which specifies the operation (read/write) for which the antenna association is valid. The associated antennae are specified inside the 'operation' tag by the 'transmitter' and 'receiver' tags. In figure 4.5, there is one association for the 'write' operation. In this association, the antenna 'top' is both transmitter and receiver.

6. library – This tag specifies the name of reader driver file and its location on the file system. A reader may have one or more driver files, each of which are specified in a 'file' tag. Thus, the 'library' tag may define one or more 'file' tags. In figure 4.5, 'libx86m4.so' and 'libmer.so' are the two driver files.

7. protocol – This tag specifies the protocol to be used by the reader to communicate with the tags. The list of protocols to be supported are specified by one or more 'type' tags within the 'protocol' tag. The protocols currently supported are PROTOCOL_ID_GEN2 (EPC Gen2 Protocol), PROTOCOL_ID_ISO14443A (ISO1443A Protocol), PROTOCOL_ID_ISO15693 (ISO15693 Protocol), PROTOCOL_ID_EPC1 (EPC1 Protocol). In figure 4.5, the supported protocol is 'PROTOCOL_ID_GEN2'.

The following are the steps carried out during the initialization and working of the readers by SmartRF.

1. The middleware configuration file is parsed and a list of readers is created which are specified in this file. This reader-list stores all the reader specific parameters such as antenna information, port information, etc., as specified in the configuration file.

2. For all the readers in the reader-list, their respective drivers are loaded from the location specified in the configuration file.

3. A connection is established with all the readers in the reader-list, whose drivers were successfully loaded. The connection with each of the reader is identified by

```
<config_info>
        <!--Specifies a particular Reader Configuration Information -->
    <reader>
            <!-- Manufacturer Name of the Reader -->
      <manufacturer_name>ABCDEF</manufacturer_name>
      <!--User defined name-->
      <username>CSE Lab</username>
      <!--Reader port configurations-->
      <port>
        <!--Type of port - (COM) (ETH) (USB)-->
        <type>ETH</type>
        <!--Location of port (COM1) (COM2) (ETH0)-->
        <location>ETH0</location>
        <!--Parameters of the port defined by various params-->
        <parameters>
            <param>
              <!--Param name (baud) (ip)-->
              <name>IP_ADDRESS</name>
              <!--Param value (IP,Baud rate)-->
              <value>172.27.22.36</value>
            </param>
        </parameters>
      </port>
          <!-- Antenna Specific Information -->
          <antennas>
           <antenna>
                <!-- Antenna ID (Used by the Reader)-->
              <id>1</id>
              <!--User defined name of the antenna-->
              <username>top</username>
              <!--Port of reader on which antenna is connected-->
              <reader_port>port1</reader_port>
           </antenna>
          </antennas>
      <!--Associations of the antennas-->
      <association>
            <!-- Operation for which association is defined-->
            <operation = "write">
              <!-- Antenna name of the tranmitter -->
              <transmitter>top</transmitter>
              <!-- Antenna name of the tranmitter -->
              <receiver>top</receiver>
            </operation>
      </association>
          <!-- The driver library of the Reader -->
          <library>
              <file>libx86m4.so</file>
              <file>libmer.so</file>
          </library>
          <!-- The protocol supported by the Reader-->
          <protocol>
              <type>PROTOCOL_ID_GEN2</type>
          </protocol>
  </reader>
</config_info>
```

Figure 4.5: SmartRF Initial Configuration File

a handle. This reader handle is then used for any subsequent communication with this reader.

4. A high level reader-object is created for every reader in the reader-list. This reader-object will be responsible for monitoring and handling all the requests to the reader from the upper layers of SmartRF. The reader-object maintains a reader-specific thread responsible for all reader communication.

5. For any reader specific call, the reader-object translates it to the HAL of the specific reader. The HAL is then responsible for dynamically loading the required function and executing it. The reply is relayed back by the HAL to the reader-object, which in-turn may forward it to the application.

Chapter 5

SmartRF Applications

RFID based auto-ID systems are being used extensively in various applications. Different applications have different sets of requirements from the RFID system. Based on these requirements the applications can be categorized into different types.

5.1 Application Types

The set of applications based on asset tracking, movement tracking and supply chain management require logic built in the application design. These kind of applications identify items based on certain user-defined events. For example, consider an RFID application to track the movement of items entering and leaving the warehouse. This warehouse has multiple gates each of which could be used to enter or leave the warehouse. In this case, the application would require a logic to determine the entry or exit of an item based on the tag read by the RFID antennae placed at these gates. One such logic could be built in the following manner. If the item is read by any of the antennae for the first time, it is considered as an entry and if a previously read tag is read the second time it is considered an exit. This would involve analysis and processing of the tag data. There are many similar applications which require complex data analysis. This processing gets complicated further considering the fact that spurious reads and read-misses are also possible in the system.

The other set of applications like item theft management and access control, do

not require significant processing or analysis of the tag data. These applications simply use the tag-ID read by the RFID reader without its previous history. For example, consider an RFID application for a departmental store which needs to record all items which have left the store. This is easily done by placing an RFID reader at the exit of the store. This reader records the items and reports them to the application. The application just displays or stores this information.

The third set of applications, such as library management and toll collection, require the tag data to be read, analyzed and written back. For example, consider an RFID application for library management. Every student may be issued an RFID tagged library card which stores information of the books borrowed by him. When a student requests to borrow a book, his card is first checked if he is permitted to borrow. Depending on the permissions, the application updates the card and the internal database or rejects the request. These applications require to support read and write operations to various memories in the tag.

5.2 Application development on SmartRF

The basic need of an RFID applications is its ability to track, locate or log a particular object at a location and then provide this information to the end user or to other enterprise applications. The applications are built using the support provided by the middleware. Thus, the flexibility and robustness in the design of the middleware plays an important role in the development and working of an application.

The hardware visualization and different features provided by SmartRF can be used effectively to build various kinds of applications. For this purpose a generic application framework[39] is designed to build applications based on SmartRF. This framework builds over the concept of data streams and channels to define application specific events. For example, an application may define an event called ENTRY_WAREHOUSE to track items entering into a warehouse. This event signifies the entry of an item into the warehouse. The definition of this event is simple as being "seen" by some channels. The events can be defined with regular expressions

where the alphabet denotes the "seen" by channels. A string accepted by this regular expression specifies that a tag data has been read in the channel order as specified in the expression. For example, consider a system with channels $C1$, $C2$, $C3$, $C4$ and $C5$ configured by an application. Two events $E1$ and $E2$ are defined as following.

$$E1 \rightarrow (C1 \mid C2)\, C3^{\star}$$

$$E2 \rightarrow C4^{+} C5$$

The event $E1$ is reported by the framework whenever a tag is read by any one of the channel $C1$ or $C2$, followed by zero or more reads by channel $C3$. Similarly the event $E2$, is reported when a tag is read one or more times by channel $C4$, followed by reading exactly once by channel $C5$. A capability to define events in such format allows an application developer to infuse intelligence into the application and thus build complex applications with little effort.
The advantages of using this framework are the following.

1. The ability of the application to specify different regular expressions, provides the flexibility to choose sequencing of tags through different sets of channels for identification of an event.

2. Improvement in the robustness and efficiency of the system, by providing the ability to handle read misses by the reader. A regular expression allows different channel strings to be accepted as an event. This feature provides an ability to handle missed reads by different channels. In our example, event $E1$ is reported when any one of the following channel strings are seen by the application.

   - $C1C3$ – Tag seen by channel $C1$ followed by $C3$.
   - $C2C3$ – Tag seen by channel $C2$ followed by $C3$.
   - $C1$ – Tag seen by only $C1$.
   - $C2$ – Tag seen by only $C2$.

Thus, even a read miss from channel $C3$ is identified as an event.

The following features of SmartRF help in simplifying the application development.

1. The ability to select one or more or even a part of the reader as a channel provides the application flexibility to define logical data stream based on the physical deployment. For example, consider the manifestation of an RFID gate (figure 5.1) using a single RFID reader. In this figure, there is a reader R0 with four antennae A0, A1, A2 and A3. Two channels $C1$, $C2$ are created, such that the data streams R0-A0, R0-A1 are a part of the channel $C1$ and R0-A2, R0-A3 are a part of the channel $C2$. This formation creates a gate, whose entry event is defined when an RFID tag moving through this gate is first seen by channel $C1$ and then by channel $C2$ (i.e. $entry \rightarrow C1C2$) and the exit event is defined when the tag is first seen by channel $C2$ and then by channel $C1$ (i.e. $exit \rightarrow C2C1$). This setup provides the application an easy way to determine the entry and exit of an object.

2. The visualization of the tag as a stream of bytes allows the application to specify all the tag related operations like filtering, writing to the tag, etc., with little effort and in a technology-independent manner. For example, consider a write data operation of the application. The following are some of the example parameters provided by the applcation to the write data function.
   Channel Handle (Value = 1)- Identifies the channel.
   Part / Full Tag ID (specified as sequence of bytes, Value = 0x112233) – Identifies a unique tag to be written irrespective of the tag protocol.
   Data (sequence of bytes, Value = 0x223344) – New ID to be written to the tag, whose original ID fully or partially matches the Tag ID (0x112233).
   The above example shows the simplicity involved in accessing the tag data as a sequence of bytes.

3. A abstraction of the hardware provided by SmartRF (data streams and channels) simplifies application development. The channels are viewed and accessed

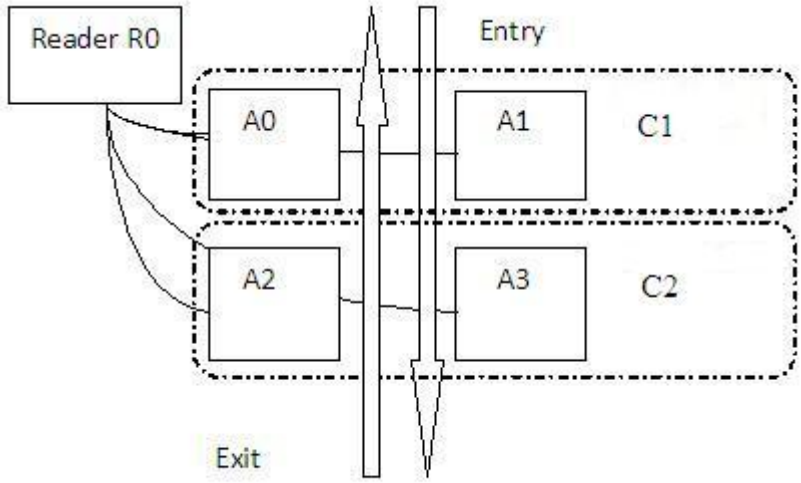


Figure 5.1: RFID Gate

in a manner similar to the file streams and are identified by a channel handle. All functions provided by SmartRF take the channel handle as argument.

5.3 Postal Bag Tracking Application

SmartRF was used to build two RFID applications – the electronic file tracking system for the department of CSE IIT Kanpur and the postal bag tracking system for speed post, India. Here, we give a brief description of the postal bag tracking system.

The department of post is an organization under the Ministry of communication and technology, India. One of the postal services called the speed post, links more than 1200 towns in India. More than 10 million articles are carried by this postal service every month. The current operation of this postal service takes place in the following manner.

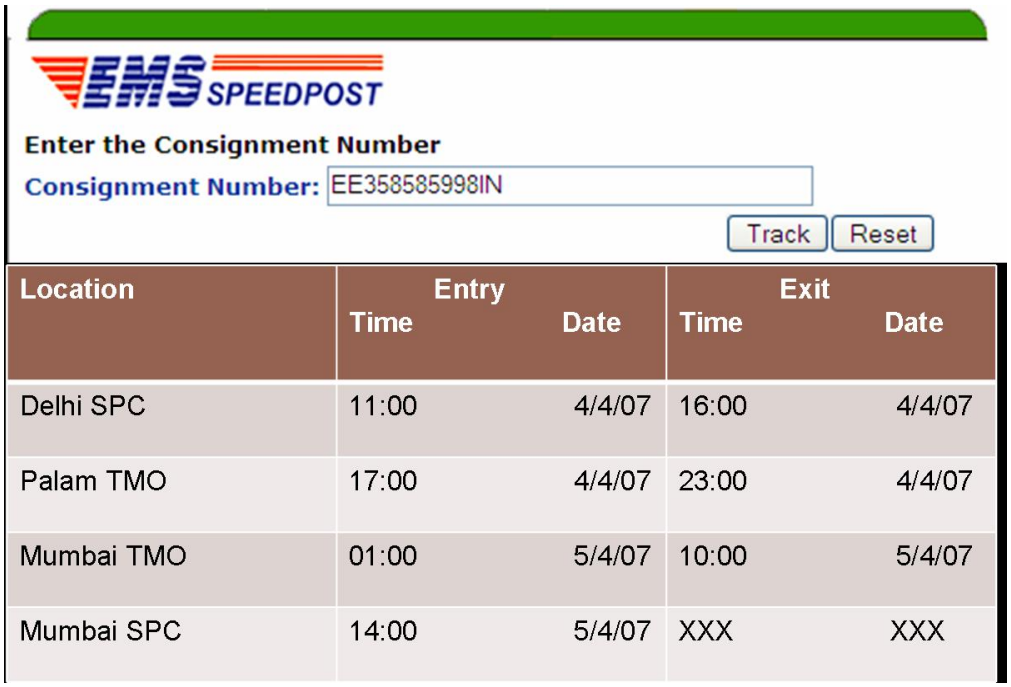
An article to be sent, is first collected at the local post office. The local post offices act as the customer-end interface and cater to small regions in the town. The articles collected at these local post offices are then carried over to a mail sorting office of the town, known as speed post center (SPC). In the mail sorting office,

all articles for a specific destination are put into a single bag. A packing list is maintained for each bag. A paper tag specifying the destination is attached to this bag. Hereon these paper tags are used to identify the bags. The bags from the SPC are then carried over to the transit mail office (TMO) for onward dispatch to the TMO covering the respective destinations. The reverse process is carried out at the destination TMO. In this existing system, the article is logged only at two points – at the source SPC and at the final delivery to the end customer. The tracking of the article is therefore very poor during its transit from the source to the destination. In cases, where the bag moves through various transit offices, it becomes impossible to track the exact location of the bag and thus the article. This incompetence may lead to chaos, if a bag is lost during the transit.

We have developed an RFID based system to track the movement of these bags. Every bag which is created at the mail sorting office is RFID tagged with the following information – bag-ID, source town, destination town, date and time of creation of the bag. Information about the articles (article-ID) contained in the bag is maintained at a central location due to the packaging list creation process. We decided to introduce the RFID tracking points at the SPCs and TMOs. These are the places, where all the bags visit and are aggregated/segregated according to the destination. The RFID readers are placed at these tracking points. At the SPC, the reader antennae are placed at the entry and exit. This helps to track the incoming and outgoing bags. Mobile RFID readers are used at the TMO for the reasons of flexibility. These mobile readers read the bags coming in or going out of the TMO. The middleware at each of these location is connected to a central database to which all the tag reads are updated periodically. The end user is provided with a web based interface to query the article. The query uses the article-bag mapping from the central database and displays the route history of the article.

An experimental prototype of this project was successfully developed and tested in the Department of Computer Science, IIT Kanpur. The figure 5.2 shows the web portal displaying the postal bag tracking information. Here the SPC and TMO

correspond to mail sorting office and mail transit points respectively.



Location	Entry		Exit	
	Time	Date	Time	Date
Delhi SPC	11:00	4/4/07	16:00	4/4/07
Palam TMO	17:00	4/4/07	23:00	4/4/07
Mumbai TMO	01:00	5/4/07	10:00	5/4/07
Mumbai SPC	14:00	5/4/07	XXX	XXX

Figure 5.2: Portal for the Bag Tacking Information

The various components involved in the development of this system are the following.

- RFID Tags - 96-bit EPC Gen2 tags from RafSec and Alien.
- RFID Readers - EPC Gen2 protocol supported readers from SmartID and Thing Magic (Mercury).
- Software System - SmartRF as the RFID middleware and the generic application framework[39] for the application development.

The experimental prototype was evaluated over a period of one month based on different hardware and software configurations. The important experiments that were carried out with respect to the hardware were testing of the system with different tags, readers, antenna placement and orientations, tag packaging materials and tag placements. The software based experiments included testing the system with and

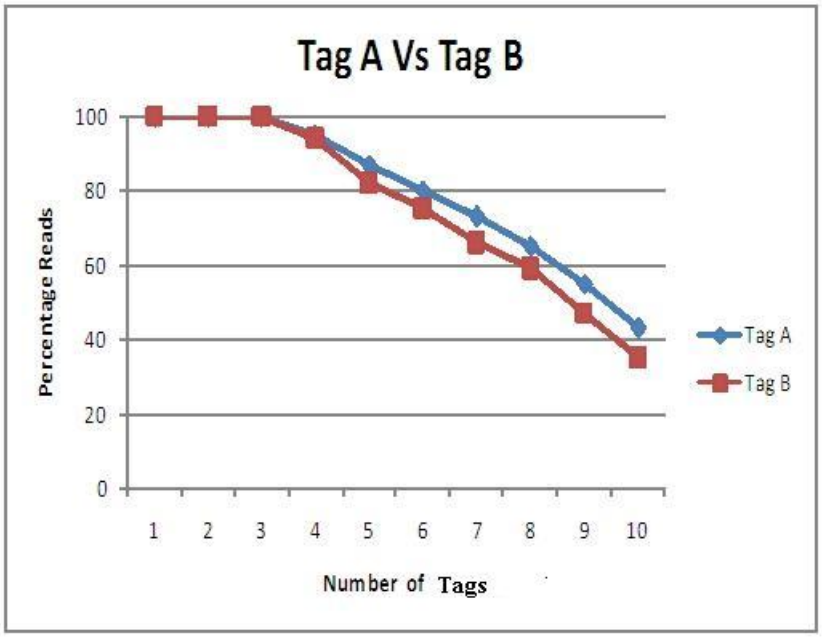


Figure 5.3: Comparison between Tag A and Tag B

without using the regular expression specification provided by the generic application framework during application development.

An important observation made during the testing of the system was that the read misses by the readers were quite significant. The tracking percentage without using regular expressions for event specification was 70 - 100%. The read percentage dropped significantly with increasing number of bags being carried together. The regular expression specification provided by the application framework was very useful and effective in capturing the events involving missed reads. The performance of the system significantly improved by including this feature.

The experiment was carried out by using tags with different antenna designs. In this experiment some tags performed better than the others, as shown in figure 5.3. The experiment results were insignificant when carried out with different UHF RFID readers. The packaging materials used for tags, played a significant role. Thicker packaging materials provided better results compared to thinner or no packaging material. This was due to the fact that, thicker materials provided the tags with an larger air gap and thus improved their performance.

Chapter 6

Performance and Results

6.1 Setup Environment

The SmartRF middleware server was run on a computer with Intel Pentium 4, 512 MB RAM, running the Windows XP operating system. This machine was put on the network and connected to the following three RFID readers.

1. SmartID reader[26] – UHF RFID reader, 4 antennas, supports serial port as the host-side communication interface.

2. Mercury 4 reader[21] – UHF RFID reader, 8 antennas, supports ethernet as the host-side communication interface.

3. NXP Pegoda reader[35] – HF RFID reader, 1 antenna, supports USB as the host-side communication interface.

The following RFID tags were placed in front of each of these readers.

1. RafSec tags[32] – UHF frequency, EPC Gen2 protocol, 96-bit UID and no user memory.

2. Alien tags[10] – UHF frequency, EPC Gen2 protocol, 96-bit UID and no user memory.

3. MiFARE Standard 1K cards[36] – HF frequency, ISO14443A protocol, 4 byte UID and 1K Byte memory.

Multiple RFID application clients were remotely connected to SmartRF via the socket interface. These clients performed operations such as to create, delete, configure, read from and write to the channels.

6.2 Performance Parameters

The aim of the experiment was to measure the following performance parameters.

1. Initial boot time of SmartRF – This is the total boot time taken by SmartRF. This time interval includes the time taken for all the initial configuration made by SmartRF such as creating a list of the readers connected to the system, initialization of the internal buffers, mutex and semaphores, and setting up a socket based interface.

2. Response time of SmartRF – The response time is the time taken by SmartRF to execute an API when requested by a client application. For example, the response time for the Create_Channel API is the time difference between the call by the client application to this API and the return code from SmartRF.

3. Memory usage of SmartRF – The memory usage includes the runtime and peak memory utilized by SmartRF during the test run. The peak memory usage is defined as the maximum memory utilized by SmartRF during the test and the runtime memory is the average memory utilized by SmartRF during multiple test runs.

6.3 Test Specifications

As discussed earlier, SmartRF was connected to different readers through various communication interfaces (Serial Port, Ethernet, USB). The test was carried out on

three computer systems. One system ran the SmartRF RFID middleware and the other two systems were the application clients.

The timing measurements of the tests were carried out using the standard APIs – QueryPerformanceTimer and QueryPerformanceFrequncy provided by the Windows operating system. These APIs provide a microsecond granularity for timing measurements.

A number of readings were taken over a week at different times with different loads to measure various test parameters. The initialization time of SmartRF was calculated by taking an average of the initialization time of SmartRF in various reader configurations. The response time of SmartRF was calculated by measuring the average time taken to execute various SmartRF APIs. The peak memory usage was calculated by measuring the maximum memory utilized by SmartRF during a test which lasted over 24 hours.

6.4 Performance Results

6.4.1 Initialization Time

The timing values below (table 6.1) are the average initialization time (in milliseconds) of SmartRF when connected with 1, 2 and 3 RFID readers.

1 Reader	2 Readers	3 Readers
318.76	663.149	1046.017

Table 6.1: Initialization time in milliseconds of SmartRF

The timing values indicate that the time taken for the initialization of SmartRF increases as the number of RFID readers connected to the system increases.

6.4.2 Response Time

The timing values shown in table 6.2 are the average response time of various APIs supported by SmartRF. The values specified here are in milliseconds.

API Name	2 Readers	3 Readers
Get_ReaderInfo()	0.0443	0.047
Create_Channel()	4.556	8.871
Read_ChannelData()	0.021	0.021

Table 6.2: Response time in milliseconds of SmartRF

Following are some of the important observations.

1. In Get_ReaderInfo(), the time taken remains constant irrespective of the number of readers connected to the system. This is because there is no data processing involved in this API call. SmartRF, executes this API by replying with the list of readers connected to the system.

2. In Create_Channel(), the overall time time taken is greater than the execution time of the other APIs. This is because this call first checks if the list of reader-antenna pairs provided by the client are a valid set or not. It then starts the read operation of the selected reader and finally creates a data structure to store and manage all channel specific information. There is also a substantial difference between the timing values, when two and three RFID readers are connected because of the increase in time for the reader-antenna pair validation.

3. In Read_ChannelData(), the execution time taken is constant irrespective of the number of readers connected to the system. This function call involves reading data from the channel buffer, applying the filter if specified and returning data to the application.

6.4.3 Runtime and Peak Memory Usage

The graph representing the runtime memory usage of SmartRF over a period of 1 hour is shown in the figure 6.1. At the start of execution the memory utilized by SmartRF is high due the processing involved in initialization of the RFID readers. After the initialization of SmartRF, the memory usage drops drastically. Then it slowly increases as more and more data is processed by SmartRF and sent to the applications. The memory usage comes down as and when the applications consume

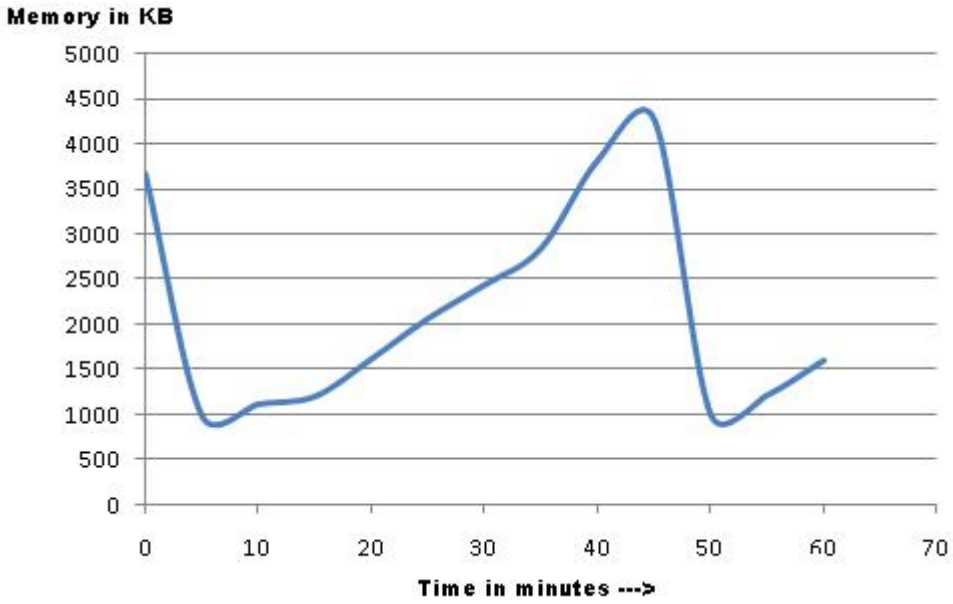


Figure 6.1: Runtime memory usage of SmartRF

the data freeing up the buffer space of the middleware. The average runtime memory utilized by SmartRF in this experimental setup was found to be 2148 KB.

For the calculation of the peak memory utilized, the load on SmartRF was increased by connecting it to 3 different clients simultaneously and increasing the number of tags in the vicinity of the readers. This setup was run continuously over a period of 24 hours. The peak memory utilized by SmartRF during this experiment was found to be 24568 KB.

Chapter 7

Conclusion

In this chapter, we present a comparison of the features and design of SmartRF with respect to the other middleware solutions. The following are some of the salient features of SmartRF.

1. Tag Abstraction – SmartRF provides the view of tag data as a stream of bytes. This abstraction gives an independence from various tag characteristics like protocols, air-interface and thus, allowing various kinds of tags to be supported by SmartRF. Many middlewares implement a protocol specific access to the tag. Thus, increasing the complexity of the applications by forcing them to understand the tag protocol specifications.

2. Reader abstraction – The reader abstraction provides a common interface to access the RFID hardware devices having different characteristics such as protocols, host-side communication interface, etc. This abstraction exposes simple functions like open, close, read, write, etc. to accomplish complex operations of the readers. This is done in SmartRF by providing a HAL wrapper which exposes the manufacturer-specific device APIs of the reader as a simple set of APIs.

3. SmartRF has provisions to handle device specific limitations such as antenna associations, which is not supported by any of the other known middlewares.

SmartRF handles these limitations internally rather than making the applications aware of it, as is done by most other middlewares. This reduces the complexity of the application and helps in a device-neutral development of the application.

4. Different middlewares available today consider a single reader as an individual source of data, which cannot be further subdivided. This prevents multiple applications to access and configure a single reader. Thus, reducing the flexibility of the system. SmartRF allows each reader-antenna pair (data stream) to be viewed as a data source. This allows applications to interact with one or more or even with a part of the reader using channel abstraction. This feature allows flexibility in the middleware configuration.

5. SmartRF presents to the application an efficient and simple method to specify the data processing specifications like the filter masks, duplicate time window. As the tags are viewed as a stream of bytes, the filter specifications are provided by the application as a set of consecutive bytes. Thus, the applications have a complete independence from various tag protocol formats. Various other middlewares use protocol specific filter specifications. For example, the Microsoft RFID Biztalk middleware uses EPCGlobal defined filter specification[27].

6. In the development of an RFID application on SmartRF, an application developer makes no assumptions regarding the underlying hardware on which the application is to run. Whereas, in most other middlewares, the application developer is exposed to the hardware internals and limitations which must be considered while developing applications. SmartRF insulates the application developer from any hardware details. This helps in a device-neutral, technology-independent design and development of the applications.

7. The modular design and open-source development of SmartRF helps in easy understanding of the middleware. It also facilitates the development and integration of newer features with little effort. The other commercial middleware

provide no such ability to extend the existing set of features of the middleware.

8. The runtime support softwares required by SmartRF are very few and are readily available. This makes SmartRF a light and portable RFID middleware. The support software required by SmartRF are the XML library required by SmartRF during initialization, MySQL database required for internal storage and the QT library for the middleware management interface. These support software integrate into SmartRF easily. Whereas, the proprietary middlewares like Biztalk (Microsoft) are heavily dependent on their own proprietary support software for their runtime operation. This makes the middleware bulky, costly and non-portable.

9. The runtime memory utilized by SmartRF under heavy load conditions is low as compared to the other middleware solutions which utilize very high memory at runtime due to the bulky support software required by them.

Future Work

We propose the following future extensions to our work.

- Performance improvement of SmartRF by reducing on the response time of the APIs and improving the memory utilization. Thorough testing of the middleware to make it more robust.

- Inclusion of additional reader configurable parameters and to develop HAL wrappers for other existing readers.

- Extend SmartRF to support EPCglobal standard and thus, making is EPC compliant.

Summary

The primary objective of this thesis was to develop an RFID middleware which allows device-neutral and technology-independent development of the RFID applications.

The developed system, 'SmartRF' is a simple, light-weight, and portable RFID middleware which provides all the generic capabilities and functionality required to be supported by a fully functional middleware. The ability of SmartRF to support multiple applications to simultaneously interact with one or more or even a part of the RFID reader provides applications flexibility and robustness compared to the other middlewares.

The postal bag tracking application is an example RFID system built with the help of SmartRF and a supporting application framework. Thus, SmartRF with its supporting application framework can be used to built complex applications with little effort.

Bibliography

[1] Pharmaceutical Product Tampering News Media Factsheet. *HDMA*, April 2004.

[2] ISO 14443-3. Identification Cards - Contactless Integrated Circuit Cards - Proximity Cards, Part 3: Initialization and Anticollision. February 2001.

[3] ISO 15693-3. Identification Cards - Contactless Integrated Circuit Cards - Vicinity Cards, Part 3: Anticollision and Transmission Protocol. April 2001.

[4] Z. Asif and M. Mandviwalla. Integrating the Supply Chain with RFID: A Technical and Business Analysis. *Communications of the Association for Information Systems*, Volume 15:393–427, 2005.

[5] P. Blythe. RFID for Road Tolling, Road-use Pricing and Vehicle Access Control. *IEE Colloquium on RFID Technology (Ref. No. 1999/123)*, Volume 1:8/1–8/16, 1999.

[6] R.W. Boss and American Library Association. *RFID Technology for Libraries*. American Library Association, 2003.

[7] J.H. Bowers and T.J. Clare. Inventory System Using Articles with RFID Tags, 1999. US Patent 5,963,134.

[8] A. Brewer, N. Sloan, and T.L. Landers. Intelligent Tracking in Manufacturing. *Journal of Intelligent Manufacturing*, 10:245–250, March 1999.

[9] Federal Trade Commision. Radio Frequency Identification: Applications and Implications for Consumers. Technical report, Federal Trade Commision, March 2005.

[10] Alien Techcology Corporation. Alien Family of EPC Gen 2 RFID Inlays, October 2007.

[11] W.J. Eradus and M.B. Jansen. Animal Identification and Monitoring. *Computers and Electronics in Agriculture*, 24:91–98, November 1999.

[12] K. Finkenzeller. *RFID handbook.* Wiley Hoboken, NJ, 2003.

[13] C. Floerkemeier, C. Roduner, and M. Lampe. RFID Application Development with the Accada Middleware Platform. *IEEE Systems Journal, Special Issue on RFID Technology*, Volume 10, December 2007.

[14] EPC Class 1 Gen2. EPCglobal Tag Data Standards Version 1.3. March 2006.

[15] R. Glidden, C. Bockorick, S. Cooper, C. Diorio, D. Dressler, V. Gutnik, C. Hagen, D. Hara, T. Hass, T. Humes, et al. Design of ultra-low-cost UHF RFID Tags for Supply Chain Applications. *Communications Magazine, IEEE*, 42(8):140–151, 2004.

[16] B. Glover and H. Bhatt. *RFID Essentials.* O'Reilly, 2006.

[17] P.M. Goodrum, M.A. McLaren, and A. Durfee. The Application of Active Radio Frequency Identification Technology for Tool Tracking on Construction Job Sites. *Automation in Construction*, 15:292–302, March 2006.

[18] EPCglobal Inc. Lower Level Reader Protocol (LLRP) 1.0.1 Specification. August 2007.

[19] EPCglobal Inc. EPC Information Services Version 1.0.1 Specification. Septermber 2007.

[20] Sun Microsystems Inc. Sun Microsystems Inc. Sun Java System RFID Software. February, 2006. http://www.sun.com/ software/ products/ rfid/ index.xml.

[21] ThingMagic Inc. Mercury 4 Datasheet. 2006.

[22] M. Karkkainen. Increasing Efficiency in the Supply Chain for Short Shelf Life Goods using RFID Tagging. *International Journal of Retail & Distribution Management*, 31:529–536, October 2003.

[23] Frank Knifsend. Oracle Fusion Middleware and Microsoft Interoperability:Addressing Enterprise-wide Needs. Technical report, January 2005.

[24] R. Koh, E.W. Schuster, I. Chackrabarti, and A. Bellman. Securing the Pharmaceutical Supply Chain. *Auto-ID Center MIT, White Paper, September*, 2003.

[25] C.J. Li, L. Liu, S.Z. Chen, C.C. Wu, C.H. Huang, and X.M. Chen. Mobile Healthcare Service System using RFID. *IEEE International Conference on Networking, Sensing and Control, 2004*, Volume 2:1014–1019, March 2004.

[26] SmartID Technology Pte Ltd. SM112 Datasheet. 2007.

[27] Microsoft. BizTalk Server 2006 Developer Productivity Study. January 2007. www.microsoft.com/biztalk/en/us/rfid.aspx.

[28] PV Nikitin, KVS Rao, SF Lam, V. Pillai, R. Martinez, and H. Heinrich. Power Reflection Coefficient Analysis for Complex Impedances in RFID Tag Design. *Microwave Theory and Techniques, IEEE Transactions on*, 53(9):2721–2725, 2005.

[29] J.H. Park, J.H. Park, and B.H. Lee. RFID Application System for Postal Logistics. *Portland International Center for Management of Engineering and Technology*, pages 2345–2352, August 2007.

[30] Christy Pettey. Gartner Says Worldwide RFID Spending to Surpass USD 3 Billion in 2010. Technical report, Gartner, Inc, December 2005.

[31] BS Prabhu, X. Su, H. Ramamurthy, C.H.I.C. Chu, and R. Gadh. WinRFID: A Middleware for the Enablement of Radiofrequency Identification (RFID)-Based Applications. *Mobile, Wireless, and Sensor Networks: Technology, Applications, and Future Directions*, 2006.

[32] UPM Raflatac. UPM Raflatac UHF Family Products, 2006.

[33] M.R. Rieback, G.N. Gaydadjiev, B. Crispo, R.F.H. Hofman, and A.S. Tanenbaum. A Platform for RFID Security and Privacy Administration. *Proceedings of the 20th conference on Large Installation System Administration Conference-Volume 20*, pages 89–102, December 2006.

[34] S.E. Sarma, S.A. Weis, and D.W. Engels. RFID Systems and Security and Privacy Implications. *Workshop on Cryptographic Hardware and Embedded Systems*, 2523:454–470, 2002.

[35] Philip Semiconductors. mifare MF RC500 Highly Integrated ISO14443A Reader IC Revison 2.0. January 2002.

[36] Philip Semiconductors. mifare Standard 4K Byte Card IC MF1 IC S70 Functional Specification. November 2002.

[37] S. Waters and S. Rahman. RFID and supply chain performance: adoption issues in the retail supply chain. *International Journal of Internet Protocol Technology*, 2(3):190–198, 2007.

[38] R. Weinstein. RFID: A Technical Overview and its Application to the Enterprise. *IT Professional*, Volume 7:27–33, May 2005.

[39] Zuber, A. Ghayal, and R. Moona. A Generic RFID Application Framework, May 2008.

Appendix A

Hardware Abstraction Layer (HAL) APIs of SmartRF

This chapter of the appendix gives a description of the set of APIs which are a part of the device driver of a RFID reader. These APIs are intended to be used by the software developer to build device driver. The developer must have knowledge of the reader functionality and a general overview of the "SmartRF – RFID Middleware". These device drivers are provided as DLLs (Windows) or shared objects (Unix).

A.1 Introduction

"SmartRF – The RFID Middleware" provides the RFID application an interface to the RFID hardware. In SmartRF, the access to the hardware is carried out by the hardware abstraction layer (HAL). All the reader devices have their device specific APIs (device drivers) provided as DLLs or shared objects. For any reader to be accessible through SmartRF, it must implement the set of APIs defined by SmartRF. In the case of SmartRF, these APIs are implemented as wrappers over the manufacturer-provided reader APIs. This chapter gives a detailed description of these APIs.

In the next section we provide the system architecture and in the final section we give a list of APIs which are to be the part of the device driver.

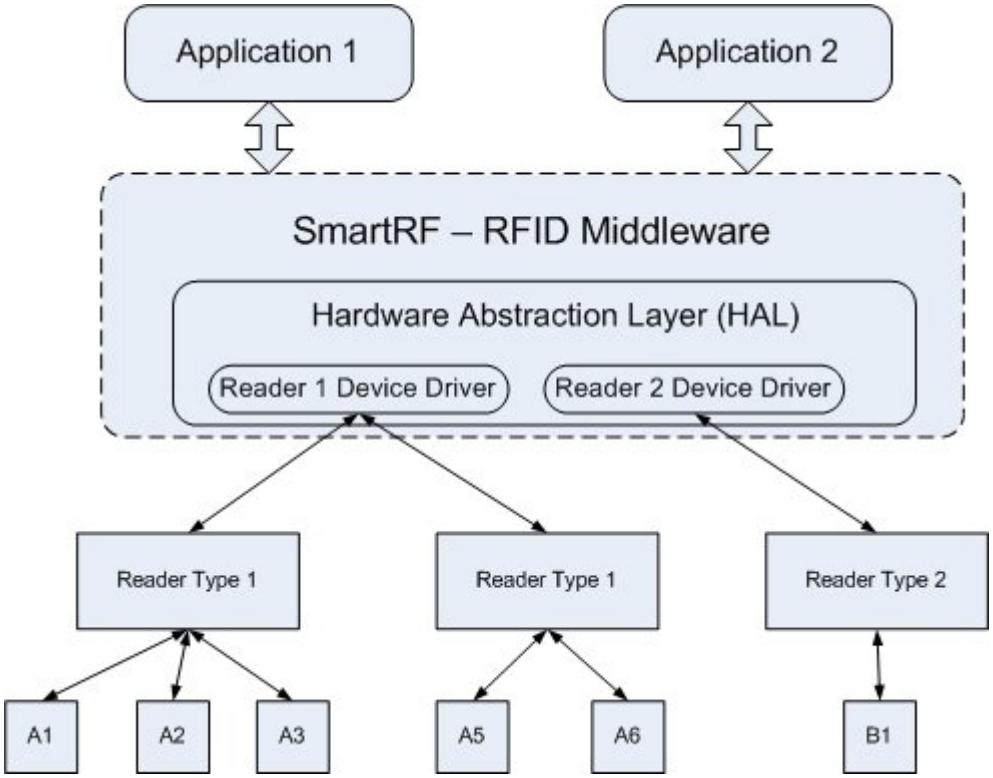


Figure A.1: RFID System Architecture (hardware)

A.2 System Architecture

The figure A.1 gives the overview of the RFID system. It also explicitly specifies the hardware interaction with the middleware. The readers have their own device drivers and these drivers are included in the middleware as a part of the hardware abstraction layer. This layer (HAL) is responsible for all the communication with the hardware. The APIs which are described here are the reader device drivers (example – Reader 1 device driver, Reader 2 device driver in the above figure A.1). Any command from the application needing an access to the RFID hardware travels through to the HAL and based on the reader handle the specific device driver HAL function is called.

A.3 API Specification

A.3.1 Open Reader

The Device_Open_Reader function is used by SmartRF to initialize a connection with the reader.

Syntax

void * Device_Open_Reader(**char** *Connection_Identifier)

Input

1. Connection_Identifier – This specifies connection type of the reader. The value for this variable must be of the form 'connection type:connection value'. For example – "IP:172.26.89.19", "COM:COM1", "USB:USB1".

Return value

Reader handle – Handle to the reader is connection is successful.
NULL – Reader is not accessible.

Description

This API checks if the specified reader exists. It returns with a handle if the reader is present. The device driver must internally have a structure to map the handle and the corresponding configuration in which the reader (Serial, USB, Ethernet) is configured.

A.3.2 Close Reader

The Device_Close_Reader function is used by SmartRF to close the connection with the reader.

Syntax

int Device_Close_Reader(**void*** Reader_Handle)

Input

1. Reader_Handle – Handle of the reader.

Return value

DEVICE_CMD_SUCCESS – Reader connection is successfully closed

DEVICE_CMD_FAILED – Reader connection cannot be closed.

Description

This API closes the connection with the specified reader. If the function call is successful the reader handle is no more valid.

A.3.3 Configure Reader

The Device_Configure_Reader function is used by SmartRF to set the reader specific parameters like the power level of operation, antenna operation specifications, etc.

Syntax

int Device_Configure_Reader(**void *** Reader_Handle, **struct Config_Params** Config)

Input

```
struct Config_Params {
    int Num_params;
    struct Reader_Params *Params ;
    }
struct Reader_params {
    int param_type;
    int param_size;
    void *param_value;
```

}

1. Reader_Handle – Handle of the reader.

2. struct Config_Params – List of reader parameters.

   a) struct Reader_Params – Array of Structure which give set of parameters to be configured.

   b) Num_Params – Number of parameters defined in the struct Reader_Params.

Return Value

DEVICE_CMD_SUCCESS – Reader is configured successfully.

DEVICE_CMD_FAILED – Reader configuration fails.

A.3.4 Start Read Operation

The Device_Start_Read function is used by SmartRF to start the read operation at the reader.

Syntax

int Device_Start_Read (**void *** Reader_Handle, **struct Read_Param** Params)

Input

```
struct Read_Param {
    int size_of_data;
    int Read_Protocol;
    }
```

1. Reader_Handle – Handle of the reader.

2. struct Read_Param Params – Structure specifies the read parameters.

   a) size_of_data – Size of data to be read by the reader

   b) Read_Protocol – Reader protocol to be used for the read operation. The following protocols can be specified.

PROTOCOL_ID_EPC1 – 0x1

PROTOCOL_ID_ISO15693 – 0x2

PROTOCOL_ID_ISO14443 – 0x4

PROTOCOL_ID_EPC0 – 0x8

PROTOCOL_ID_GEN2 – 0x10

Return Value

DEVICE_CMD_SUCCESS – Reader read starts successfully.

DEVICE_CMD_FAILED – Reader fails to start read operation.

Note

The size_of_data parameter in the Read_Param structure defines the following.

size_of_data = unique ID (based on the protocol) + user data.

Based on the size of data to be read and the protocol supported, the API decides the size of unique ID and user data, and internally calls functions to access these memory banks separately. For example, if the size_of_data = 20 bytes and Read_Protocol = PROTOCOL_ID_GEN2 then, unique ID = 12 bytes and user data = 8 bytes.

A.3.5 Get Read Data

The function Device_Get_Data is used by SmartRF to read data from the driver buffer.

Syntax

int Device_Get_Data(**void *** Reader_Handle, **struct Get_Read_Info** *Params)

Input

```
struct Get_Read_Info {
    int data_valid;
```

int antenna_id;

char *data;

int Protocol_Read;

}

1. Reader_Handle – Handle of the reader.
2. struct Get_Params *Params
   a) data_valid – Indicates the validity of the data.
   b) antenna_id – Antenna ID of the antenna which has read the data.
   c) data – Pointer in memory which is used to return the read tag data.
   d) Protocol_Read – Reader protocol used to communicate with the tag.

Return Value

Size of data – Read is successful.

DEVICE_CMD_FAILED – Operation fails.

A.3.6 Stop Reader Read

The function Device_Stop_Read is used by SmartRF to stop the read operation of the reader.

Syntax

int Device_Stop_Read(**void *** Reader_Handle)

Input

1. Reader_Handle – Handle identifying the reader.

Return Value

DEVICE_CMD_SUCCESS – Reader read operation is successfully stopped.

DEVICE_CMD_FAILED – Reader read operation cannot be stopped.

A.3.7 Number of tags available

The Device_Num_Tags function returns the number of tags available in the reader buffer which are not yet read by SmartRF.

Syntax

int Device_Num_Tags(**void *** Reader_Handle)

Input

1. Reader_Handle – Handle of the reader.

Return Value

Number of tags – Function call executed successfully.
DEVICE_CMD_FAILED – Function fails.

A.3.8 Write Data

The Device_Write function is used by SmartRF to write data to the specified tag.

Syntax

int Device_Write (**void *** Reader_Handle, **struct Write_Params *** Params)

Input

```
struct Read_Params {
    char *Tag_ID ;
    char *Tag_Data;
    int data_length;
    int Protocol_Write;
    }
```

1. Reader_Handle – Handle of the reader.

2. struct Write_Params *Params –

   a) Tag_ID – Tag ID of the tag which is to be written to.

   b) Tag_Data – Data which is to be written to the tag.

   c) data_length – Length of data to be written to the tag (length of Tag_Data).

   data_length = Tag ID (UID) + user data.

   if(data_length < UID) – Tag_Data value is the new UID. Bytes of old UID replaced with Tag_Data.

   if(data_length > UID) – Tag_Data value is the new UID + user data.

   if(data_length = UID) – Tag_Data value is the new UID.

   d) Protocol – Reader protocol to be used for writing to the tag.

Return Value

DEVICE_CMD_SUCCESS – Write to tag is successful.
DEVICE_CMD_FAILED – Write to tag has failed.

A.3.9 Select Antenna

The Device_Select_Antenna function is used by SmartRF to power up the antennas of the reader.

Syntax

int Device_Select_Antenna(**void *** Reader_Handle, **int** Selected_Antennas[], **int** Num_Antennas)

Input

1. Reader_Handle – Handle of the reader.
2. Selected_Antennas – Array of antenna numbers for a specific reader device which are to be powered on.
3. Num_Antennas – Number of valid entries in the 'Selected_Antennas' array.

Return Value

DEVICE_CMD_SUCCESS –Antennas successfully selected.

DEVICE_CMD_FAILED – Antennas selection failed.

A.3.10 Set Associations

The Device_Set_Association function is used by SmartRF to set associations between antennas of a reader.

Syntax

int Device_Set_Association(**void *** Reader_Handle, **struct Associations** Associate)

Input

```
struct Associations {
    int Transmitter_Antennas[];
    int Num_Transmitter;
    int Receiver_Antennas[];
    int Num_Receiver;
    }
```

1. Reader_Handle – Handle of the reader

2. struct Associations – Structure specifies the association information.

   a) Transmitter_Antenna[] – Array of antenna numbers to be selected as transmitters for an association.

   b) Num_Transmitter – Number of valid entries in the Transmitter_Antenna array.

   c) Receiver_Antenna[] – Array of antenna numbers to be selected as receivers

for an association.

d) Num_ Receiver – Number of valid entries in the Receiver _Antenna array.

Return Value

DEVICE_CMD_SUCCESS – Antennae are successfully associated

DEVICE_CMD_FAILED – Antennae association failed.

A.3.11 Check Reader Connectivity

The Device_Check _Connected function is used by SmartRF to check if the reader device is connected to the system.

Syntax

int Device_Check _Connected(**void *** Reader_Handle)

Input

1. Reader_Handle – Handle of the reader.

Return value

DEVICE_CMD_SUCCESS –Device is connected.

DEVICE_CMD_FAILED – Device is not connected.

Appendix B

Application Abstraction Layer (AAL) APIs of SmartRF

B.1 Introduction

'SmartRF' is an RFID middleware which provides the application a device-neutral interface to access the hardware. The applications running over SmartRF perform a number of operations, some of which are the following.

- Reading and writing data to the tags .
- Configuring the middleware data processing features such as filtering, duplicate time window, etc.
- Add and delete readers to the middleware.

SmartRF provides the application with APIs to perform all the above specified operations. These APIs are provided as a part of the application abstraction layer (AAL) of SmartRF. In this chapter of the appendix, we provide a list of all the APIs which are provided to the application by SmartRF.

SmartRF is implemented as a daemon which interfaces multiple applications with multiple readers. The application and SmartRF are separate processes which may be running on different systems and communicating via the socket interface. The

following is the general flow of an RFID application when it needs the services of SmartRF.

1. Connect to the SmartRF daemon.
2. Use services (APIs provided) by SmartRF to configure/access devices.
3. Close the connection with SmartRF.

In the next section, we give a the system architecture and in the final section we provide a list of all the APIs.

B.2 System Architecture

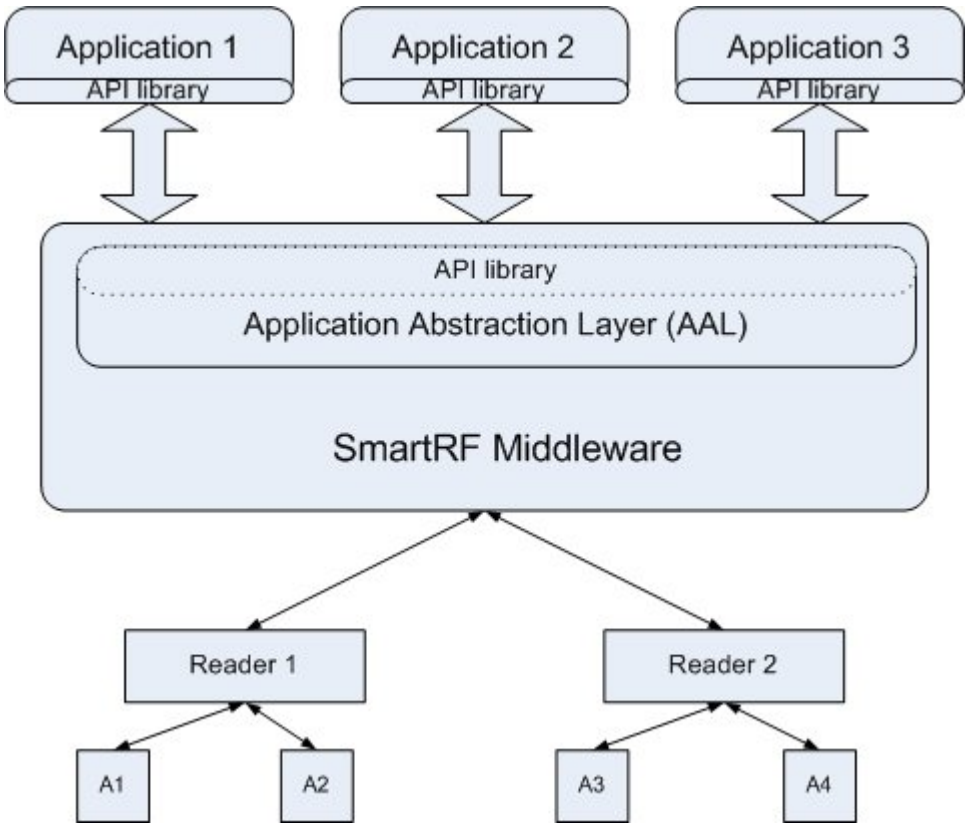


Figure B.1: RFID System Architecture (applications)

The figure B.1 gives an overview of the RFID system architecture. Multiple applications communicate with SmartRF using the available APIs. The application ab-

straction layer (AAL) of SmartRF is responsible for handling all the communication with the applications. The AAL interprets the call from the application and passes on the required information to the lower layers of SmartRF, which are responsible for accessing and configuring the hardware.

B.3 API Specification

B.3.1 Connect to the middleware

The connect_app function is used by the application to establish a connection with SmartRF.

Syntax

int connect_app(**string** ip_address_port, **Connection_descriptor** *mwConnection)

Input

1. ip_address_port – IP address and the port of the middleware server. This value is specified as IP_address:port_value. For example '172.24.24.19:11002'.
2. mwConnection – Address of the variable which holds the information about the type of connection made with the middleware.

Return Value

SMARTRF_SUCCESS – Application connects to SmartRF.
SMARTRF_FAIL – Application fails to connect to SmartRF.

B.3.2 Get reader list

The Get_Reader_List function is used by the application to get the list of readers connected to the SmartRF.

Syntax

int Get_Reader_List(**Connection_descriptor** mwConnection, **Reader_Info** **Reader_list
)

Input

```
struct Reader_Info{
    string Manufacturer_Name;
    string Userdefined_Name;
    string Port_Type;
    string Port_Location;
    int Num_Port_Params;
    Generic_Params *Port_Params;
    int Num_Antennas;
    string Antenna_Name[];
    int Num_Protocols;
    string Protocol[];
}
struct Generic_Params {
    ssize_t Param_Type;
    int Param_Size;
    void *Param_Value;
}
```

1. mwConnection – the variable which holds the information about the type of connection made with the middleware.

2. Reader_list – Pointer to memory where the list of readers is returned by the function call.
   a) Manufacturer_Name – Manufacturer name of the reader.
   b) Userdefined_Name – User defined name of the reader.

c) Port_Type – Host system port information. (IP / USB / COM).

d) Port_Location – Location of the port where reader is connected. (IP / COM1 / COM2).

e) Num_Port_Params – Number of port parameters.

f) Port_Params – Port parameter values. The parameters are BAUD_RATE, IP_ADDRESS, etc.

g) Num_Antennas – Number of entries in the 'Antenna_Name' array.

h) Antenna_Name – Array of user names of all the antennae connected to the reader.

i) Num_Protocol – Number of entries in the 'Protocol' array.

j) Protocol – Array of all the protocols supported by the reader.

Return Value

Number of readers – Function is successful.

SMARTRF_FAIL – Function fails.

B.3.3 Create Channel

The Create_Channel function is used by the application to create a channel.

Syntax

int Create_Channel(**Connection_descriptor** mwConnection , **Channel_Create** Channel)

Input

```
struct Channel_Create {
    int Operation_Type;
    string Channel_Name;
    int Num_Streams;
    Data_Stream *Streams;
```

}

struct Data_Stream {

string Reader_Name;

string Antenna_Name;

}

1. mwConnection – Variable to store the connection type with SmartRF.
2. **struct** Channel_Create – Structure specifying the channel information.
   a)Operation_Type – Operation for which the channel is to be used (read / write).
   b)Channel_Name – Name of the channel to be created.
   c)Num_Streams – Number of entries in the Stream array.
   d)Stream – Array of reader-antenna pairs which are to be a part of the channel.

Return Value

Channel Handle – Channel created successfully.

SMARTRF_FAIL – Channel creation fails.

B.3.4 Change channel configuration

The channel_config function is used by the application to change the existing channel configuration. Several parameters are available for the channel, each of which may be changed through this function call. Table 4.1 provides a list of all configurable channel parameters.

Syntax

int channel_config(**int** Channel_Handle, **Channel_Param** Params)

Input

struct Channel_Param {

int Num_Params;

list<Generic_Params *> Ch_Param;

}

1. Channel_Handle – Handle of the channel that is to be configured.
2. **struct** Channel_Param – Structure with a list of parameters to be configured.
   a) Num_Params – Number of entries in the 'Ch_Param' list.
   b) Ch_Param – List of parameters to be configured.

Return Value

SMARTRF_SUCCESS – Channel parameters configured.

SMARTRF_FAIL – Channel parameter configuration failed.

B.3.5 Get channel configuration

The Read_Config function is used by the application to get the values of parameters for a channel from SmartRF. Table 4.1 provides a list of all channel parameters.

Syntax

int Read_Config(**int** Channel_Handle, **Channel_Info** **Config_info)

Input

struct Channel_Info{

int Channel_Handle;

int Operation_Type;

string Channel_Name;

int Num_Streams;

Data_Stream Streams[];

Channel_Param Params;

}

1. Channel_Handle – Handle of the channel.

2. Config_info – Pointer to memory location which returns the channel parameters.

   a)Channel_Handle – Handle of the channel.

   b)Operation_Type – Operation type of the channel (read/write).

   c)Channel_Name – User defined name of the channel.

   d)Num_Streams – Number of entries in the Data_Stream array.

   e) Data_Stream – Array of the reader-antenna pairs.

   f) Params – Array of channel parameters.

Return Value

SMARTRF_SUCCESS – Channel parameters are successfully returned.
SMARTRF_FAIL – Function fails.

B.3.6 Destroy Channel

The Destroy_Channel function is used by the application to delete the specified channel from SmartRF.

Syntax

int Destroy_Channel(**int** Channel_handle)

Input

1. Channel_handle – Handle of the channel that is to be destroyed.

Return Value

SMARTRF_SUCCESS – Channel successfully destroyed.
SMARTRF_FAIL – Channel delete failed.

B.3.7 Read data

The Read_data function is used by the application to read data from SmartRF.

Syntax

int Read_data(**int** Channel_Handle, unsigned char *buffer, **ssize_t *** time_data**,**
int read_type)

Input

1. Channel_Handle - Handle of the channel.
2. buffer – Pointer to memory where the data is read by the function.
3. time_data – Pointer to memory where the time is read by the function.
4. read_type – Type of read operation (blocking / non-blocking).

Return Value

SMARTRF_FAIL – Read was unsuccessful.
Number of bytes read by SmartRF – Read was successful.

B.3.8 Write data

The Write_data function is used by the application to write data to a selected tag.

Syntax

int Write_data(**int** Channel_Handle, **unsigned char** *partial_tag_id, **int** tag_id_length,
unsigned char *data, **int** data_length)

Input

1. Channel_Handle- - Handle of the channel
2. partial_tag_id – Partial tag IS which identifies the tag which is to be written to.

3. tag_id_length – Length of partial_tag_id.

4. data – Data that is to be written to the tag.

5. data_length - Length of the data to be written.

Return Value

SMARTRF_SUCCESS – Write to tag is successful
SMARTRF_FAIL – Write to tag fails.

B.3.9 Exit application

The Exit_Application function is used by the application to close its connection with SmartRF

Syntax

void Exit_Application(**Connection_descriptor** mwConnection)

Input

1. mwConnection – Connection descriptor that identifies the application.

Return Value

void