November 20th, 2011

Dr. Andrew Rawicz
School of Engineering Science
Simon Fraser University
Burnaby, British Columbia
V5A 1S6

Re: ENSC 440 Design Specification for an RFID Smart Fridge

Dear Dr. Rawicz:

The document attached outlines the design specification for Cyber-Flux Innovations' RFID Smart Fridge. Our team is designing a smart refrigerator which will keep track of its contents and provide useful information to customers through a website and mobile phone application. Our design specification document applies to the proof-of-concept prototype we are currently developing. Design improvements for future models of the RFID Smart Fridge are discussed, but will not be implemented in this stage of development.

This document will direct the development and testing process of the prototype. Our team is comprised of four talented undergraduate students from Simon Fraser University:
Lead Integration Manager: Damir Jungic (Electronics stream),
Lead Hardware Design: Mitchell Joblin (Biomedical stream),
Lead Software Design: Steven Verner (Computer stream), and
Lead Project: Renato Pagliara (Biomedical stream).

If you have any questions or concerns regarding our functional specifications, please do not hesitate to contact us at rfridgeidtech@googlegroups.com.
Sincerely,

Renato Pagliara

Renato Pagliara
Project Lead
Cyber-Flux Innovations

Enclosure: *RFID Smart Fridge Functional Specification*

RFID SMART FRIDGE

DESIGN SPECIFICATION

Project Team:

Damir Jungic
Mitchell Joblin
Steve Verner
Renato Pagliara

Contact Person:

Renato Pagliara
rpa13@sfu.ca

Submitted to:

Dr. Andrew Rawicz – ENSC 440
Mike Sjoerdsma – ENSC 305
School of Engineering Science
Simon Fraser University

Issued Date:

November 20th, 2011

EXECUTIVE SUMMARY

The design specification of the RFID Smart Fridge provides a set of detailed descriptions of the design and development of our proof-of-concept prototype system. The design specifications in this document apply solely to the proof-of-concept prototype. While the specifications in this document might deviate from the features of the system as described in the document *RFID Smart Fridge Functional Specification* (Pagliara, Jungic, Joblin, & Verner, 2011), we have added features which were not previously described which address key issues of power and performance. We have integrated a temperature and humidity sensor into the in-Fridge hardware to provide users with the ability to monitor the conditions inside the fridge in real-time. At the same time, the addition of a reel switch located on the fridge's door allows us to maintain the power consumption of the system to a minimum while ensuring we keep basic network functions running.

This document outlines the design of the RFID Smart Fridge and provides justification for our design decisions. The RFID Smart Fridge System development has been divided into four modules as well as system integration: In-Fridge Hardware, Backend Server, Server Features and Smartphone App. The in-Fridge hardware located within the fridge reads RFID tags inside of the fridge using a RFID reader/writer unit. The reading of the fridge's contents is triggered by the closing of the fridge door. The information from the tags is then sent to the server through a Local Area Network (LAN) using a WiFi module. This process, together with the temperature/humidity readings from the sensor, is coordinated through an Arduino microcontroller.

The Backend Server saves tag information received from the Hardware Unit. The server then extracts the barcode information from each tag and runs it through a database to obtain the products' name and nutritional information. Once the information from the products has been obtained, the user can observe the contents of their fridge through either a web browser or through their iPhone. As well, users can automatically generate a grocery list based on the history of their fridge's contents and consumption prediction algorithms. Finally, nutrition models, calculated from the nutritional information retrieved from a product catalogue database, will allow the users to quantify the level of nutrition for each item inside the fridge. The iPhone will leverage all the processing power of the backend server and will work as a web client which will help costumers navigate through the many features offered by the Smart Fridge System.

Design decisions, including selection of hardware, are justified throughout this document. General software program process flow is also included. Further, a description of the test plan for the system and its subcomponents is included. Finally, a discussion about the safety of RFID technology is presented to ensure this critical aspect of product development is not overlooked.

The expected completion date for the RFID Smart Fridge project is December 06, 2011.

ACRONYMS & ABBREVIATIONS

1-D	**One-Dimensional**
App	**Application**
HTML	**HyperText Markup Language**
HTTP	**HyperText Transfer Protocol**
ISO	**International Organization for Standardization**
LAN	**Local Area Network**
LED	**Light-emitting Diode**
MB	**Megabyte**
RFID	**Radio Frequency Identification**
Temp	**Temperature**
UID	**Unique Hardware Identification**
UML	**Unified Modeling Language**
UPC	**Universal Product code**
WiFi	**Wireless-Fidelity**

GLOSSARY

Radio frequency identification system: An automatic identification and data capture system comprised of one or more reader/interrogators and one or more transponders in which data transfer is achieved by means of suitably modulated inductive or radiating electromagnetic carriers (AIM Global, 2010).

Transponder: An electronic TRANSmitter/resPONDER, commonly referred to as a Tag [same as above].

Unique Hardware Identification (UID): throughout this document, we will define UID as an eight hexadecimal characters long number which uniquely identifies a transponder.

Universal Product Code (UPC or EAN): barcode symbology widely used in North America, for tracking trade items in stores. Its most common form, the UPC-A, consists of 12 numerical digits, which are uniquely assigned to each trade item. Along with the related EAN barcode, the UPC is the only barcode allowed for scanning trade items at the point of sale, per GS1 standards (GS1, 2011).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

The RFID Smart Fridge is a smart home appliance which besides functioning as a regular fridge, it keeps track of items tagged with RFID transponders placed inside the fridge and provides customers with extra information about their products, their nutritional facts and their fridge's contents history. This document describes the technical details behind the Smart Fridge system. The information has been divided into the system's four main components: in-Fridge hardware, backend server, server features and smartphone app. The document also includes a test plan for the integration of all four components into the system.

Scope

This document specifies the design of the RFID Smart Fridge system and explains to what degree have the functional requirements as described in *RFID Smart Fridge Functional Specification* (Pagliara, Jungic, Joblin, & Verner, 2011) been met. At the same time, new features not listed in the previously mentioned document have been added to the system and are explained in detail in this document. The design specification encompasses the requirements for the proof-of-concept system and a partial set of requirements for a production model.

Intended Audience

The design specification is intended for use by the members of Cyber-Flux Innovations.
Lead engineers will refer to the specifications as guidelines for the development of the system but in case they deem it necessary they will modify these specifications to ensure all requirements are met in the final product. This product will also be used during the testing phase to ensure to confirm the correct behaviour of the Smart Fridge system.

2 SYSTEM SPECIFICATIONS

The RFID Smart Fridge will consist of a modified fridge that detects items being put in or taken out via their RFID tag (using an RFID reader). The reader then sends the tag information to a server via a LAN where the information will be processed and made available to the user through a simple mobile application and website as seen in the diagram below. The server will process this large amount of information and make it available in a simple, convenient, useful manner through its interfaces to give the user a list of the fridge's current contents, a list of foods which will expire soon, an automatically generated grocery list based on the user's eating habits and some simple and relevant nutritional information about the fridge's contents and recent history.

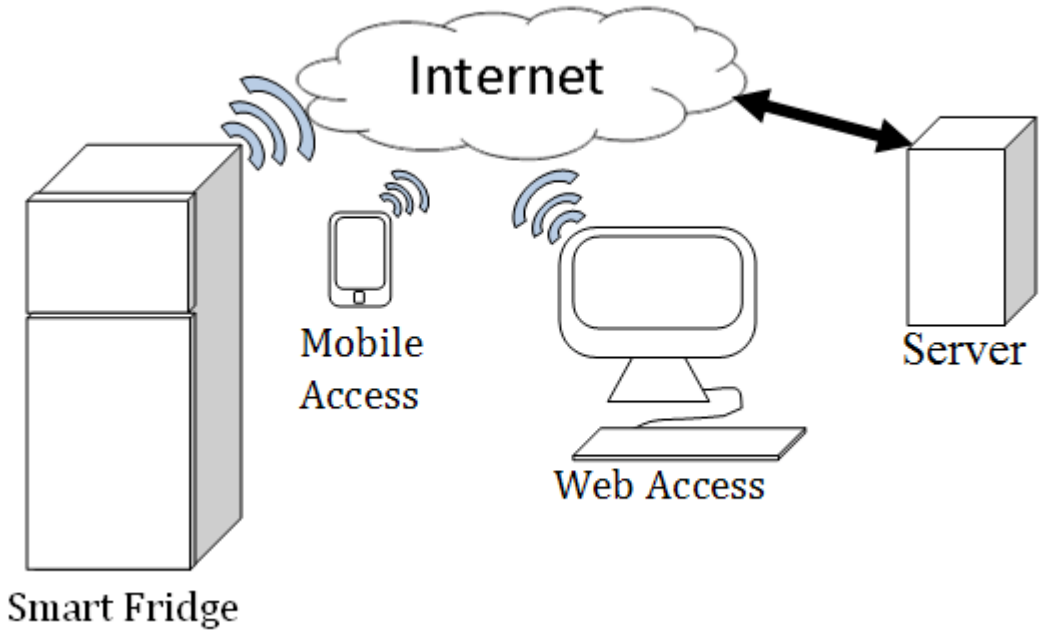

Figure 1: System Overview

RFID Technology

Radio Frequency Identification (RFID), as its name implies, makes use of radio waves to transfer information from a transponder (in our case, a tag) to a reader, with the purpose of identifying the transponder and thus the object it is attached to. An RFID system consists of three elements: an antenna, a transceiver with a decoder to interpret the data and a transponder that has been programmed with information. The antenna emits radio-frequency signals in a relatively short range. The RF radiation provides a mean of communicating with the transponder (the RFID tag) and provides the RFID tag with the power necessary to communicate (in the case of passive transponders).

Thanks to the transmission of power through the RF-signals, RFID tags do not need to contain batteries, and remain usable for years. When an RFID tag passes through the field of the scanning antenna, it detects the activation signal from the antenna. That "wakes up" the RFID chip, and it transmits the information on its microchip to be picked up by the scanning antenna.

RFID tags can be either active or passive. Active RFID tags have their own power source; this means that the reader can be much farther away and still get the signal. Passive RFID tags, however, do not require batteries, and can be much smaller and have a virtually unlimited life span. RFID tags can be read in a wide variety of circumstances, where barcodes or other optically read technologies are useless. The tag need not be on the surface of the object (and is therefore not subject to wear). The read time is typically less than 100 milliseconds. Depending on the RFID reader, large numbers of tags can be read at once. For simpler cheaper readers, items should be read item by item (technovelgy.com, 2010).

RFID Technology and Safety

The safety of underlying technologies plays a key role in high-level design decisions. Indeed, the very plausibility of the Smart Fridge System requires that the underlying technology not harm people or the food that the system comes into contact with. It is necessary to consider two primary safety issues: does RFID technology interfere with food and does it interfere with pacemakers or similar electrical medical devices worn users.

It has been observed that RFID tags operating at low frequencies can affect the operation of pacemakers (US Food and Drug Administration, 2010). However, the larger read range of high-frequency readers make them preferable for the Smart Fridge System. Higher-frequency signals are routinely filtered by worn medical devices, and therefore pose no known risk to device behaviour. The Smart Fridge System will therefore not harm users who rely on worn or implanted medical devices.

Studies observing the effects of Non-Ionizing Fields, such as those created by RFID readers, only demonstrate short-term effects (such as heating) (RFID 4SME and the European Union, 2008). Further, it is worth noting that an Electro-Magnetic Field (EMF) is only present within the fridge while the RFID reader is reading tags. Tags are only read after the fridge door is closed. Further, the reading the contents of the entire fridge only takes seconds. The exposure of the food within the fridge to an EMF is therefore extremely minimal. The Smart Fridge System therefore is not harmful to the food with which it comes into contact with.

3 OVERALL SYSTEM DESIGN

A high-level overview of the Smart Fridge System is presented in Figure 2. From the point of view of costumers, the RFID Smart Fridge solution has three main components: the In-Fridge Hardware, the Backend Server and the Smartphone App. Even though the product catalogue is not part of the server (it is a standalone software service available through the web), from costumer's point of view it appears as if the server is performing the services which belong to the product catalogue.

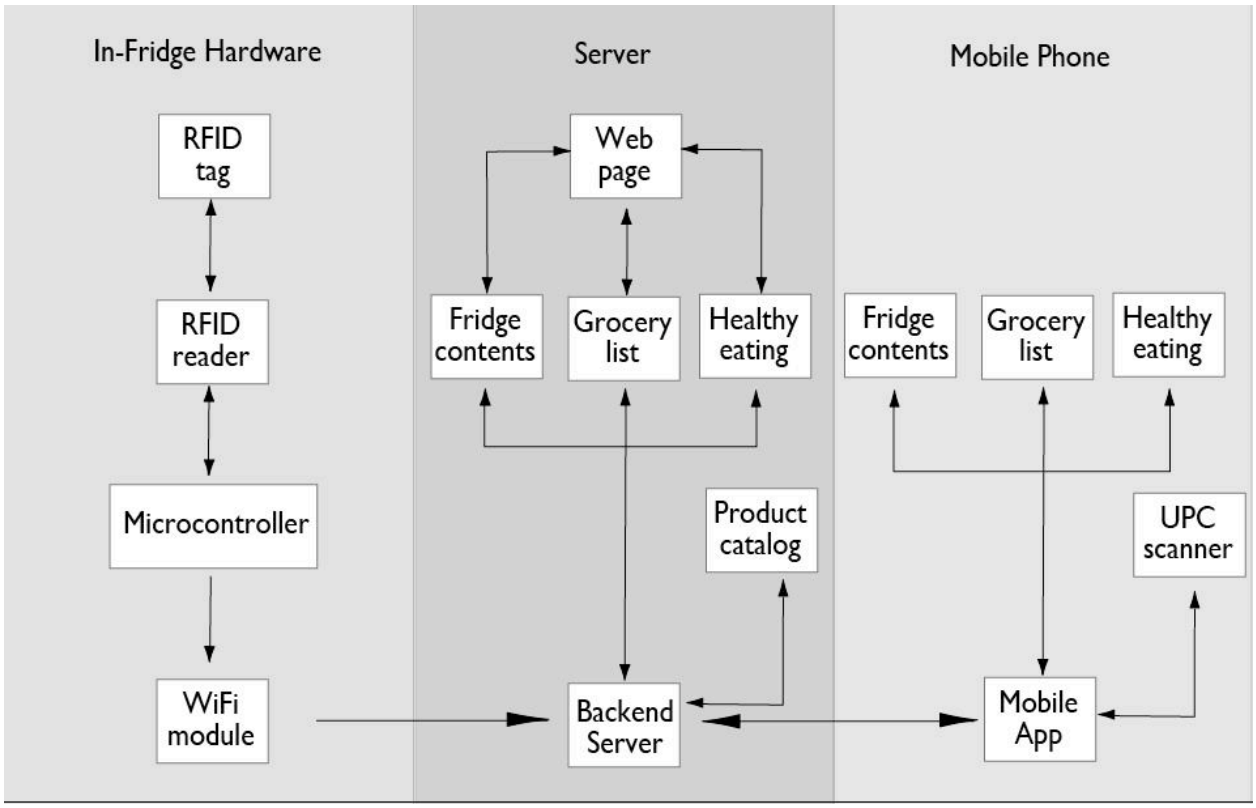

Figure 2: System Overview

The In-Fridge Hardware is installed within the fridge and is in charge of obtaining information about the products placed inside the fridge through the RFID tags. The server takes that information and, upon request from the user through the webpage, allows users to check the contents of the fridge, generate grocery lists and check the nutritional value of present and past products within the fridge. In the same way, the user can access these three features through the Smartphone App. Finally, the Smartphone App offers a fourth feature where costumers will be able to enter items into the system that do not have an RFID tag by either entering the name of the product (for leftovers) or by scanning the UPC barcode (for products which do not have an RFID tag) and attaching a RFID read/write tag to the product.

The system thus, has two general cases:

Products with a built-in RFID tag:

In this case the information flows in a linear fashion through the system. When the door of the fridge closes, the in-Fridge hardware scans the fridge for RFID tags. After reading all the tags within the fridge, the information is enclosed in a message and sent to the server.

Products without a RFID tag:

In this case the user starts the flow of information by entering the name or scanning the UPC barcode of the item through the Smartphone App. The user then attaches a RFID tag to the product/leftover and inserts it into the fridge. The flow of information from there onwards is the same as in the first case.

This document has been divided into sections dedicated to the explanation of the main subsystems within the Smart Fridge solution. The design of the In-Fridge Hardware, Backend Server, Product Catalogue, Software System and the Smartphone App components of the system are explained in the following sections. In addition, a section dedicated to communication will describe the interactions of the various subsystems.

4 IN-FRIDGE HARDWARE

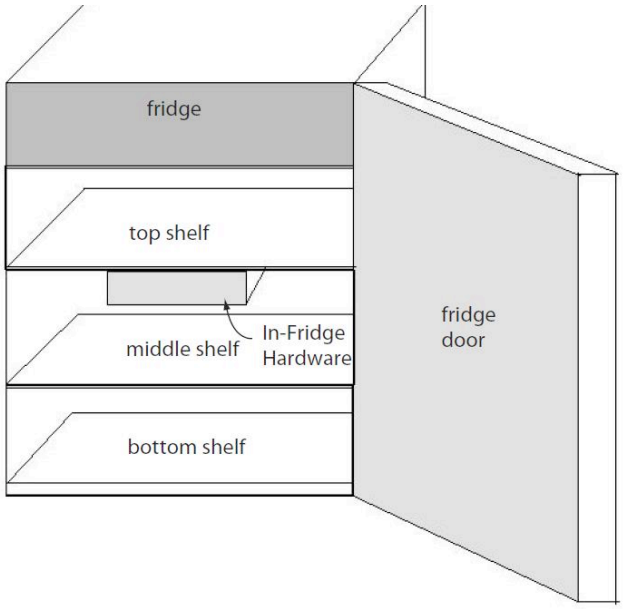

Figure 3: In-Fridge Hardware location within the fridge

Figure 3 illustrates the location of the In-Fridge Hardware within the fridge. This location was selected to maximize the reading range and thus to be able to read any product located inside the fridge. The In-Fridge Hardware unit consists of a RFID reader/writer, a microcontroller, a WiFi module and a temperature/humidity sensor.

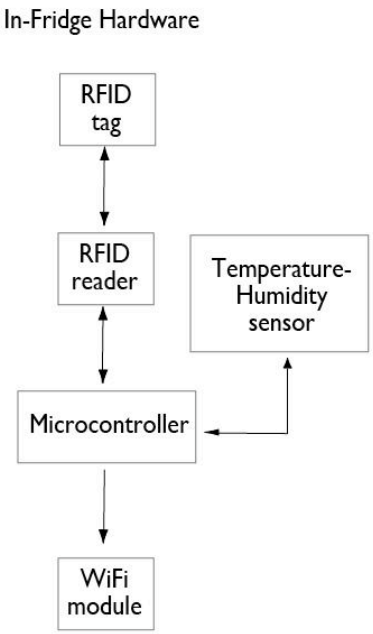

Figure 4: In-Fridge Hardware subsystem diagram

RFID reader/writer:

Since the reader/writer unit is an integral part of the project, there were several requirements taken into account, such as:

- Radio waves frequency
- Communication range
- Data Transfer Rate
- Power Consumption
- RFID protocol compatibility
- Price

RFID technology is used in 3 main frequencies, Low Frequency LF, High Frequency HF, and UltraHigh Frequency UHF. It follows that all other requirements for the RFID reader/writer unit depend on the frequency of the reader as shown in Table 1. Given the requirements of the project and the small budget, it was decided that the Smart Fridge prototype would use the APSX-RW 310 RFID reader/writer unit. This unit has a range of approximately 25cm above and below the built-in antenna (rectangular shape, dimensions: 25cm x 10cm) and approximately 8cm to the sides with credit card size RFID tags. The unit communicates to external devices through a UART Serial port. The fourth column in Table 1 shows the characteristics of the unit. As well, the prototype Smart Fridge uses the ISO15693 RFID communications protocol, which is supported by the APSX reader/writer unit chosen.

Table 1: RFID commercially available reader/writer units and their characteristics (Ward & van Kranenburg, 2006)

	LOW FREQUENCY	HIGH FREQUENCY	ULTRA HIGH FREQUENCY	APSX-RW 310
Frequency Range	30-300kHz	3-30MHz	300MHz-3GHz	NA
Typical working frequency	125-134kHz	13.56MHz	433 MHz or 865 – 956MHz 2.45 GHz	13.56kHz
Approximate Communication Range	Less than 0.5m	Up to 1.5m	433 MHz = up to 100 metres 865-956 MHz = 0.5 to 5 metres	0.1-0.25m
Typical Data Transfer Rate	Less than 1kbit/s	Approximately 25kbits/s	433–956 = 30 kbit/s 2.45 =100 kbit/s	19.2kbits/s
Power consumption	Low	Low	High	20mA/5V
Characteristics	Short-range, low data transfer rate, penetrates water but not metal.	Higher ranges, reasonable data rate, penetrates water but not metal.	Long ranges, high data transfer rate, concurrent read of <100 items, cannot penetrates water or metals.	Short-range, reasonable data transfer rate, penetrates water but not metal.
Typical Use	Animal ID, car immobilizer	Smart-Labels, contact less travel cards, Access & Security	Specialist animal tracking, Logistics	NA
Price	Lowest	Low	High	Affordable

Microcontroller

The microcontroller is in charge of sending command to the RFID reader/writer unit, temporarily store information of RFID tags, preparing the messages to be sent to the backend server and communicating with the WiFi module and temperature/humidity sensor. The microcontroller-backend server communication details are provided in the communication section of this document.

The Arduino UNO microcontroller was selected for the Smart Fridge prototype due to its simplicity of use, libraries support and previous experience of some of the members of the team with the device. As well, the Arduino UNO comes with receive and transmit UART serial ports while allowing for the creation of extra software serial ports if necessary. The Arduino UNO microcontroller requires 7-12V and 40mA current per I/O pin.

WiFi Module

In order to transmit the information from the RFID tags to the server, it is necessary to connect the microcontroller to the server through a LAN using WiFi. WiFi was selected over Bluetooth, XBee, ANT and other wireless protocols since it is the native language of the server, it has the longest range and it provides extra features that allow for better control and management over the communication. For this prototype, the AsyncLabs WiFi shield offered good quality at a low price while providing basic transmission control protocol support. This unit communicates to the microcontroller through UART serial. The WiFi module requires 250µA of current and 5V to function.

Temperature/Humidity Sensor

Given that the main feature of any fridge system is the ability to maintain food products at a low temperature to prolong their edible life, the Smart Fridge prototype monitors the temperature and humidity inside the fridge and reports the values to the server. The SHT15 temperature and humidity sensor from Sensirion comes already calibrated and provides extremely high precision temperature and humidity readings which makes it ideal for the Smart Fridge prototype. This unit is also low power (30 µW/2.4-3.6V), has a temperature range of -40°C to 123.8°C (±0.3°C at 25°C) and relative humidity (RH) range of 0-100% RH (±2%RH). This sensor communicates to external devices through Inter-IC (I2C).

ISO15693 RFID Protocol

The selected protocol for communication between the RFID tags, the RFID reader/writer and the microcontroller is the ISO15693. This protocol is the most widely used RFID protocol and it can be found in most RFID products. This protocol requires external devices attempting to communicate to RFID tags to do so through bytes of data representing the request for information to the device. The general format of the ISO15693 command is the following:

Start Of File (SOF) (N bytes)	Flags (1 byte)	Command (1 byte)	Parameters (0-N bytes)	Checksum CRC (2 bytes)	End Of File (EOF) (N bytes)

Figure 5: General ISO15693 request

Where the flags represent different parameters that deal with the data transfer rate and the transmission of the command to one specific RFID transponder or to all transponders within the read range, the command represents the requested action and the parameters is the information necessary to process the requested action.

The SOF and EOF protocol is usually dependant on the RFID reader/writer unit selected. In the case of the APSX-RW 310, the SOF is equal to the number of bytes enclosed in the message (not counting the SOF byte) and the EOF is not specified (this means, the message is considered complete after the checksum bytes are sent).

Microcontroller-RFID reader/writer Communication

The communication between the microcontroller and the RFID reader/writer unit was designed for efficiency and easy debugging. Numerous functions were implemented within the microcontroller code

which allows the microcontroller to transmit the necessary bytes to the RFID reader/writer to request for specific information from a specific RFID transponder within the reading range.

At the most simple level, a function has the task to send one request in the ISO15693 format to the RFID reader/writer unit. The function is also in charge of processing the response from the RFID unit and to save the relevant information into global variables that can then be accessed by other functions for data processing.

For example, if the microcontroller needs to request the RFID reader/writer unit to read the UID of all transponders within the reading range, then the microcontroller calls a set of functions that perform simple specific tasks: first, the microcontroller calls a function which reads one UID of a transponder present in the read range. The UID information is then stored in a variable "tagUID". Then, this variable is passed as a function parameter to another function which sets that transponder in a sleeping mode in which the transponder does not react anymore to requests. Finally, this process is repeated until no transponders react to the request.

Power

The voltage and current necessary to power the RFID reader/writer unit, the WiFi module, and the temperature/humidity sensor can all be provided by the Arduino UNO microcontroller. The microcontroller requires a steady power supply of 7-12V. The microcontroller will draw power from the power supply of the fridge. In order to do this, a 120V-to-12V regulator will be placed in parallel with the power supply for the fridge.

To reduce the amount of power consumed by the in-Fridge hardware, a reel switch and a magnet will be placed on the fridge's door. This switch will control the power to the RFID reader/writer unit. The switch will connect a 5V output pin to one of the interrupt pins, both in the microcontroller. The interrupt is set to be triggered when the signal changes from low to high. Hence, when the fridge door and the switch open, the interrupt will not report a change. The interrupt will be triggered when the door closes and the signal goes to 5V, thus signalling the microcontroller that the item set within the fridge might have changed. The microcontroller will then read all the items within its range and report the new item set to the server.

Casing

The in-Fridge hardware will be placed within a plastic case. The case will protect the electronics inside it from damage without any interfere with the radio waves. The temperature/humidity sensor will be placed on the opposite side to the microcontroller and WiFi modules to reduce any possible temperature and humidity error resulting from the heat emanated by the electronic devices.

5 BACKEND SERVER

The Backend Server was built using the Java Programming Language. Java was chosen due to its popularity, platform independence, and the extensive availability of libraries, APIs and documentation. A full-featured programming language was chosen over a server-side scripting language due to the speed of computation and flexibility of the former. The system often needs to deal with large data sets (ie. a full history of products that have been inside the user's fridge), something that the interpreted nature of server-side scripting does not handle well.

Object Hierarchy

Items within the fridge, such as a can of Coke or a container of leftover pizza, are represented within the Backend Server as ItemType objects. The ItemType class is abstract – nothing can be *just* an ItemType. The fields of an ItemType object and their meanings are presented in Table 2.

Table 2: Description of ItemType Fields

Field	Description
id	hardware ID of the RFID tag attached to this item
expiry_date	the date at which the item will expire
fridgeTimeStamps.entry_time	the time the item entered the fridge
fridgeTimeStamps.exit_time	the time at which the item left the fridge (for good)
consumingTimeStamps	a list of entry time & exit time pairs that describe every time the item was taken out of the fridge and put back in.

There are two classes that extend ItemType: LeftoverItemType and ProductItemType.

A LeftoverItemType object represents a container that was RFID tagged by a user. In addition to all the fields of an ItemType, a LeftoverItemType contains a "name" field that serves as a user-specified label of the contents of the container.

Objects of type ProductItemType represent items with UPC codes. ProductItemType objects have an additional field called "upc" which specifies their UPC. It is worth noting that the "upc" field is of type String, despite always having a numeric value. This is necessary because leading zeros are significant when specifying a UPC (read below about serialization to see why a xml-primitive type is necessary instead of a more sophisticated object type). For example, two cans of Coke constitute two objects of type ProductItemType – both with the same value of the "upc" field.

The Smart Fridge System attempts to pull information (including brand, description, a picture, nutritional information, etc) from the Product Catalogue (discussed below) whenever a new ProductItemType object is added to the InFridgeSet. The information pulled from the catalogue is identical for all products with the specified UPC. Therefore, to avoid storing multiple copies of the same information this information is not attached to the ProductItemType, but is rather turned into a new object of type ProductType.

A ProductType is a UPC barcode and all of its associated information, including description, brand, picture URL, and potentially nutritional information. This object is intended to contain all the information that doesn't change from one *instance* of a product to another. For example, two cartons of milk with the same UPC code can have different expiry dates, which is why this information is contained in the ItemType (each carton will have its own ItemType object). However, both cartons have the same brand name, description, nutritional information, and so on. Both the "info" (including brand name, description etc) and the nutritional information are stored as hashtables, since the UPC catalog service we are interfacing with (SimpleUPC) can return different amounts of information depending on the UPC that is queried.

The remaining objects serve as containers for those described above. There are three containers for ItemType objects:

- InFridgeSet – contains all items that are currently in the fridge.
- HistorySet – contains all items that were once in the fridge (it has yet to be decided if this set will only contain items for a certain period of time). Once an item is removed from the InFridgeSet, it is added to the HistorySet. If an item is taken out of the fridge and then put back, it leaves the HistorySet and reenters the InFridgeSet.

There is also a ProductCache that stores the results received from SimpleUPC. In this way, the number of queries sent to the service is reduced and requests can be responded to more quickly. By separating this store of information, sophisticated caching techniques become possible in order to reduce user wait time and server bandwidth.

A FridgeSystem object contains one of each of these three sets: an InFridgeSet, HistorySet and ProductCache. This is the complete description of a system.

All the object relationships explained thus far are presented in the UML diagram in Figure 6.

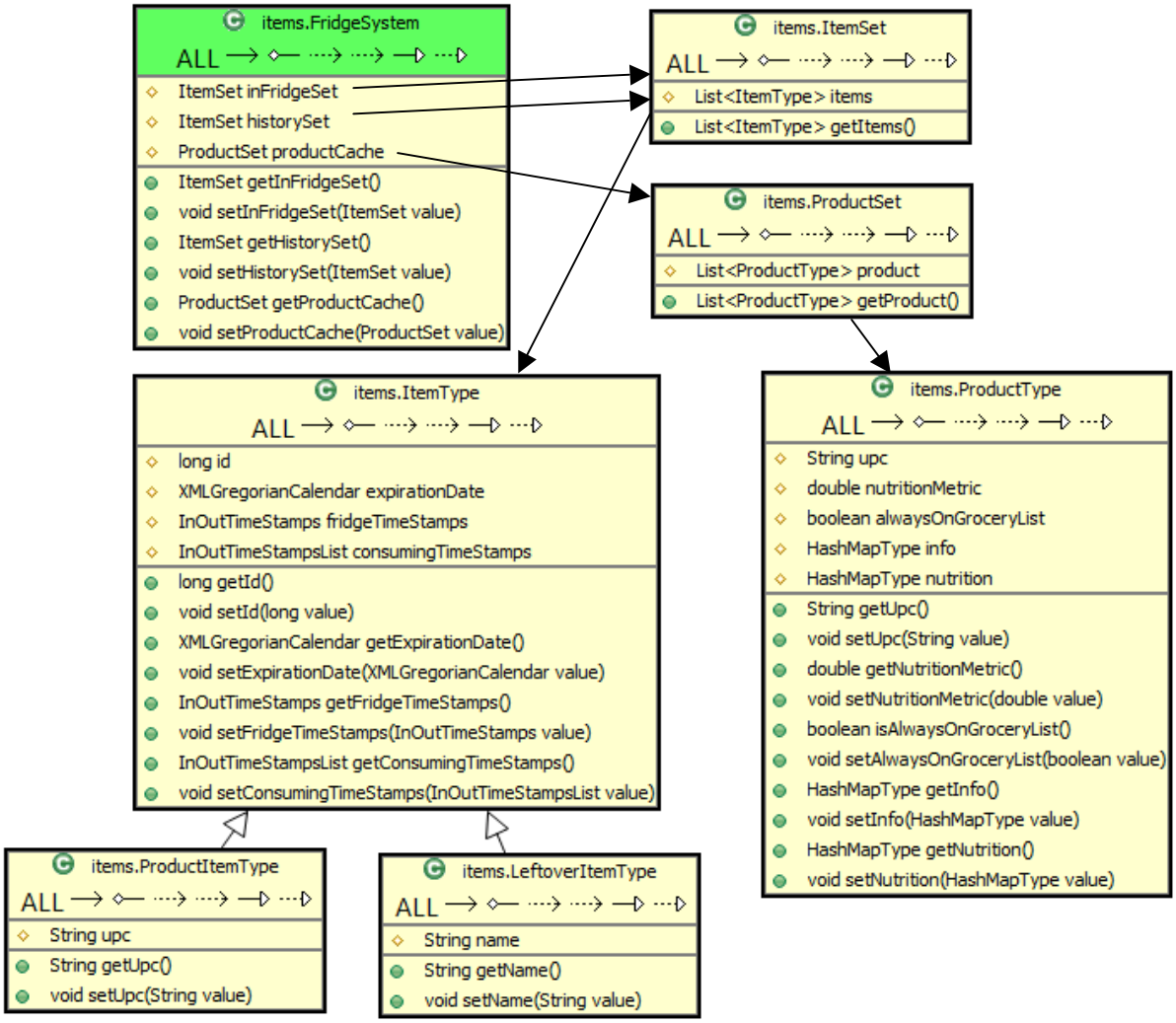

Figure 6: UML Diagram of Core Backend Server Classes

Adding and Removing Items

One of the core functions that the Backend Server must perform is the adding and removing of ItemType objects to and from the InFridgeSet and the HistorySet. The flowchart presented in FIGURE described

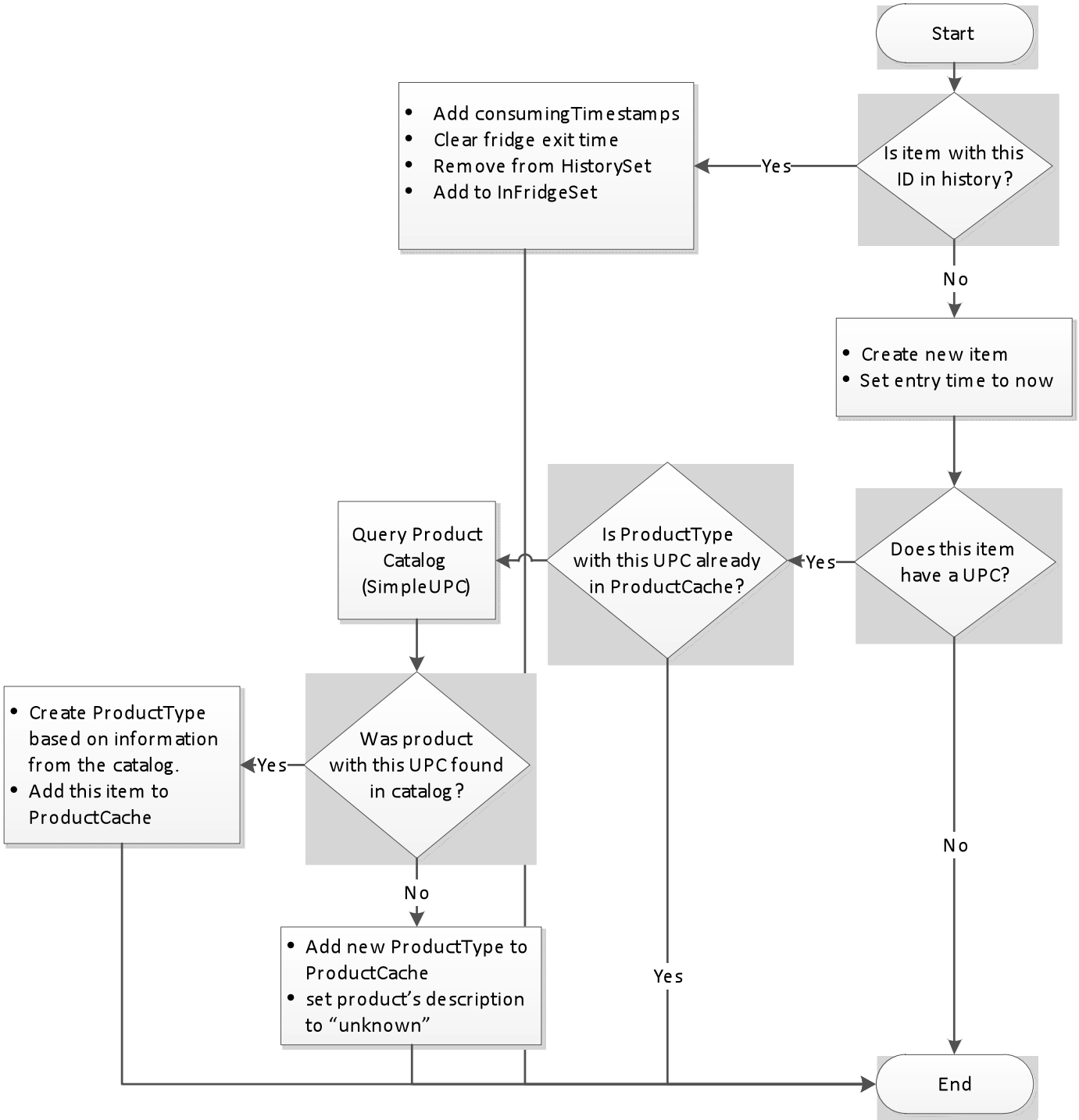

Figure 7: Adding to the InFridgeSet Process Flow

When an enters the fridge, it is necessary to determine whether this is the item's first time entering the fridge, or if it is being put back after having a portion consumed. If it the system's first time seeing an item with this UPC (if there is no entry in the ProductCache with this UPC), it is necessary to pull information from the Product Catalogue. If the product is not available through the Product Catalogue, an entry with description "unknown" is added to the ProductCache.

Note that events which trigger HTML generation are not included in the above Figure 7, as it only pertains to item tracking.

The process of removing items from the InFridgeSet is much simpler than adding them. First, the item must have its `exit_date` set to the current time. The item is then removed from the `List<ItemType>` within the InFridgeSet and added to the `List<ItemType>` within the HistorySet.

Serialization

It is frequently necessary to transmit complete objects – this is the role of serialization. A rigorous mapping is made between object fields and a binary format (in our case, XML text). In this way, objects utilized by the system can be written to files or transmitted easily and completely.

The Backend Server utilizes the Java-XML Binding (JAXB) in order to serialize objects. A viable alternative to XML-based serialization is JavaScript Object Notation (JSON). Both formats are popular, with XML being more established but JSON receiving increasing amount of attention. Ultimately, either would have server our purposes. JAXB was chosen due to its mature and well established codebase and the ability of Extensible Stylesheet Language Transformation (XSLT) to transform XML into HTML (GlassFish, 2011). It is also worth mentioning that Database Object Binding is also a potential means of persistent storage, though it is in general more cumbersome.

The entire serialization mechanism utilized by the Smart Fridge System is presented in Figure 8.

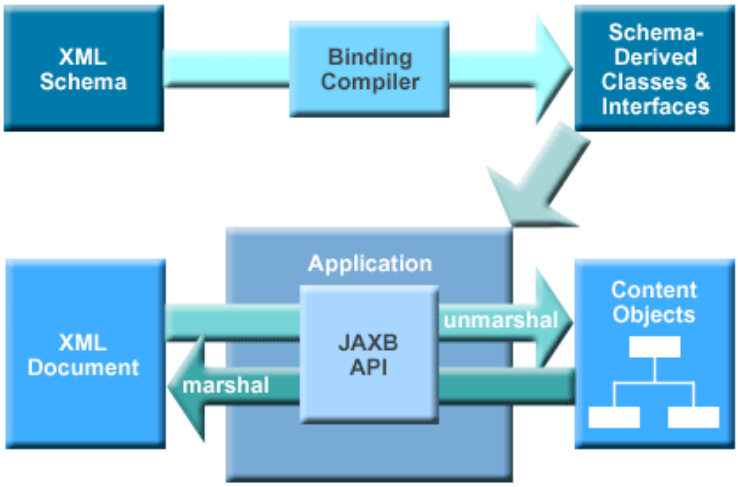

Figure 8: Java XML Binding (JAXB) Workflow

An XML schema is written to specify the fields of objects. A binding compiler then transforms this XML schema into java source files that define the same objects as described within the schema(s). These objects can then be used in any Java code, identically to any other objects. Objects generated by this mechanism can then be transformed into XML through a process called "marshaling." For example, the

XML-representation of an object can be written to a file through a call to the `marshal()` function. These saved objects can then be retrieved by a running program through the `unmarshal()` function, which receives XML data as an argument.

All of the classes described above were generated by JAXB in the manner described above, including: ItemType, ProductItemType, LeftoverItemType, ProductType, InFridgeSet, HistorySet, FridgeSystem.

The Smart Fridge System relies on XML serialization for several critical purposes:

- Crash recovery
- Data transmission
- HTML generation
- Testing

Firstly, retrieving the state of the system through reading files provides a mechanism for crash recovery. On startup, the Smart Fridge System loads objects from XML files. During execution, these files are periodically written to, as triggered by a configurable timer. In this way, minimal data is lost should the server be terminated for whatever reason, and operation is automatically resumed whenever the server is relaunched.

Secondly, when a mobile client requests information (say, the set of all items inside the fridge), the requested objects only need to be marshaled onto the OutputStream of the response object. Further, the client knows the exact form of the response - the XML schemas that were initially used to generate the Java source files of the transmitted objects specify the format.

Next, XSLT has the ability to transform the data present in an XML document into HTML. XSLT is essentially a programming language that is executed using XML documents as input, performing some sort of transformation. For the purposes of the Smart Fridge System, we transform the XML that represents the InFridgeSet (as defined above) into HTML. This HTML is then made available through the web server.

Finally, object serialization allows for much simpler and more thorough testing. Because the system state is loaded from an XML file, it is possible to manually edit this file in order to create any desirable test state. Further, a client was created in order to add ItemType and ProductType objects to the server. In this way, the client is able to verify all the functionality of the server, including Item/Product management and network functionality.

For an exact description of the XML representation of objects, see the XML schemas within the "`FridgeSystem/schema/`" folder.

Client's Responsibilities

Note that the client is completely responsible for how data is presented to the user, and that the Backend Server only transmits raw data. If we again consider two cartons of milk with the same UPC code, the user should likely be informed that they have two cartons of milk. That is, these items should

not be to distinct entries the list of items that are within the user's fridge, but rather a single entry that indicates two instances of the same product. Further still, the client might still wish to indicate that the milks might have to different expiry dates. Clearly the presentation decisions of the client need not directly reflect the object model that the Backend Server relies on.

Note also that the client needs to bind the data between a ProductItemType and a ProductType as required. For example, if the client has a particular ProductItemType that is in the fridge and would like to show its image, the client must find the associated ProductType (that is, the ProductType with the same UPC as the ProductItemType), and look inside the "info" hashtable for the value with key "`imageThumbURL`." The FridgeSystem.xml file can be used as a reference when determining which key-value pairs are available in the "info" and "nutrition" hashtables, but note this information isn't necessarily available (since it is pulled from SimpleUPC).

6 PRODUCT CATALOG

The Product Catalogue is responsible for providing brand names, descriptions, pictures, and nutritional information. The Backend Server queries the Product Catalogue by UPC, and stores the resulting information within the ProductCache (see Object Hierarchy). Our current implementation of the Smart Fridge System utilizes SimpleUPC as a Product Catalogue service (SimpleUPC, 2010).

SimpleUPC was chosen as the Product Catalogue for our current implementation. They granted free access for the full four-month development period. The service provides an API for accessing the following information by UPC:

- Manufacturer
- Brand
- Description
- Size and Weight
- Image
- Ingredients
- Nutritional Information

Further, it is possible to search the SimpleUPC database by keyword. While this feature might be used to suggest alternatives to products, the matching has absolutely no context and therefore is not very useful in this usage. For example, searching "yogurt" might return products such as yogurt-covered pretzels and yogurt covered pretzels. Clearly these are not viable alternatives for a particular brand of yogurt. When implementing the system, we used this search functionality in order to simulate a history of products, all of which have nutritional information. We were then able to run our algorithms to ensure that they produced useful results.

The system is queried via either XML or JSON, and responses are received in the same format. The Smart Fridge System queries SimpleUPC through XML, for consistency with other implementation decisions (namely, serialization). An example request to SimpleUPC is presented in Listing 1, and an example response is presented in Listing 2.

```
<request>
    <auth>API Key</auth>
    <method>FetchProductByUPC</method>
    <params>
        <upc>073731001059</upc>
    <params>
</request>
```

Listing 1: Simple UPC Request Example

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
    <success>true</success>
    <usedExternal>false</usedExternal>
    <result>
        <brand>Mission</brand>
         <manufacturer><![CDATA[Mission Foods]]></manufacturer>
         <container>Box</container>
         <description><![CDATA[18 Ct Taco Shells]]></description>
         <size><![CDATA[6.75]]></size>
         <category><![CDATA[Mexican Dinner Mixes, Tortillas, Corn Shells]]></category>
         <units>Oz</units>
         <upc>073731001059</upc>
         <ProductHasImage>true</ProductHasImage>
         <ProductHasNutritionFacts>true</ProductHasNutritionFacts>
    </result >
</response >
```

Listing 2: SimpleUPC Response Example

It is worth noting that the Smart Fridge System was designed not to rely on a particular Product Catalogue. An interface exists from which a ProductType objects is received. In this way, code written to interface with any other service that provides the necessary information can be utilized as long as it adheres to this interface. Further, since the system only ever pulls information from the ProductCache, there is a layer of abstraction between information used by the system and where the information is coming from. This design decision is significant since the nutritional features provided by the Smart Fridge System rely heavily on the Product Catalogue. Should a more comprehensive or feature-rich catalogue be available, the Smart Fridge System needs only to be adjusted slightly in order to leverage the superior catalogue.

7 COMMUNICATION

Several components of the Smart Fridge System exchange information. Each of these communication exchanges is presented in Figure 9 and expanded in the following subsections.

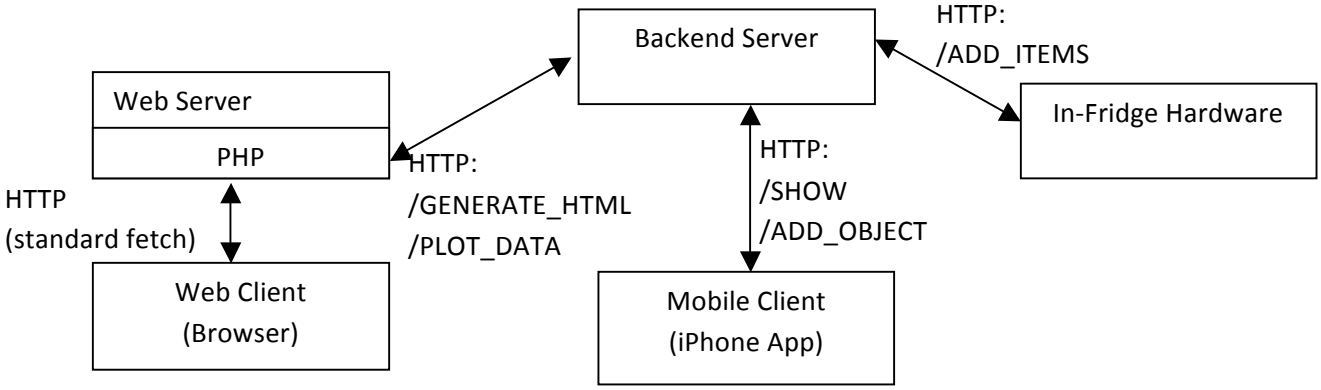

Figure 9: Communication Overview

As observer in Figure 9, all inter-component communication goes through the Backend Server. The services provided by the Backend Server are therefore first discussed, and then how the various components utilize these services is expanded upon in subsequent sections.

It is important to note that the Hypertext Transfer Protocol (HTTP) plays two distinct roles within the Smart Fridge System: a standard role when web clients send requests to the web server to fetch HTML files, and a custom purpose when the In-Fridge Hardware or mobile app send and request information to and from the server. Our custom usage of the HTTP protocol is specified in the following subsections. For the definition of HTTP, see (Fielding, Gettys, & Mogul, 1999).

Backend Services

All communication to the Backend Server is facilitated by HTTP. Clients send requests to different Universal Resource Identifiers (URIs) in order to perform different tasks. For HTTP, URIs have the following format:

```
http://<server_ip>:<server_port>/<path>?<query>
```

HTTP was chosen as a message format for many reasons. Firstly, testing is facilitated the ability to use any browser as a client. Further, HTTP is a very established protocol for which there are libraries on essentially all modern platforms. An arbitrary message format could have easily been decided on, but the ubiquity of HTTP libraries (and as a protocol in general) left little reason for a custom-designed protocol.

For the purposes of the Backend Server, the `<path>` portion of the URI can have one of the values described in Table 3 (note that these values are **case sensitive**).

Table 3: Paths Made Available by Server

Path	Description
SHOW	Retrieves information from the server in XML format as defined in `FridgeServer/schema`. The query portion of the URI (discussed below) specifies what information is returned, but most commonly it will be the current

	contents of the fridge. No content is necessary within the request body.
SAVE	Allows clients to cause the server to save its current state. A filename can optionally be included within the request body, otherwise a default filename is used.
ADD_OBJECT	Allows clients to add fully-formed XML objects to the server, and is therefore used primarily for testing. Currently supported objects include ItemTypes and ProductTypes. The object to be added must appear within the HTTP request body in the XML format specified in `FridgeServer/schema`. The query portion of the URI allows users to specify which set objects will be added to, as discussed below.
ADD_ITEMS	The "real" path used by the In-Fridge Hardware unit. The HTTP message body includes XML that defines the set of all items currently in the fridge. The server then compares this incoming set to the "current" set in order to determine which items have been removed and added.
GENERATE_HTML	Used to generate the HTML representation of information that is made accessible through the web server. This HTML is generated through XSL Transformations. The exact HTML that can be generated is presented in Table 4.
PLOT_DATA	Provides a set of points within the response body, having the format: "[[0,0],[1,1],…]". The data available through this path is used by the web server and mobile app to present graphical information. The exact information that is available is presented in Table 4.

If a request with a path other than those specified in Table 3 is received, the server sends a HTTP Response Header with code 404 (Not Found). If an entry does not have a response specified in Table 3, the server will respond with only a Response Header with code 200 (indicate everything is OK).

The `<query>` portion of the URI contains a set of key-value parts separated by '&' symbols. For example, a valid request to the Backend Server would use a URI such as:

```
http://127.1.1.8:8278/SHOW?target=SYSTEM
```

The response body will then have all the information about the current Fridge-System in XML format (including InFridgeSet, HistorySet and ProductCache). However, clients will typically not need a description of the entire fridge system, but rather just the InFridgeSet. Complete descriptions of the key-value pairs that are require or possible for each URI path is presented in Table 4.

Table 4: Paths and their Queries

Path	Query	Description
SHOW	target={SYSTEM | INFRIDGE | HISTORY}	**Optional.** Specifies which set is returned in the body of the response.
SAVE	*None*	

ADD_OBJECT	target={INFRIDGE | HISTORY} **case sensitive**	**Mandatory if ItemType is being added, unnecessary for ProductTypes.** Specifies which set the object to be added should go into.
ADD_ITEMS	*None*	
GENERATE_HTML	target={INFRIDGE | HISTORY | TIME_SENSITIVE}	**Mandatory.** Specifies what HTML will be generated. If INFRIDGE or HISTORY is supplied, the HTML representation of that set is generated into the standard HTML (`web/html`) folder. If the value is et to TIME_SENSITIVE, the HTML that represents the expiring soon, already expired and recently removed items are all generated.
PLOT_DATA	target= {EXPIRY_HISTORY | <upc_value>}	**Mandatory.** Supplying the EXPIRY_HISTORY target returns x-values representing months and y-values representing the number of items which expired before being removed for that month. If the target supplied is a UPC number, the x-values correspond to months and the y-values indicate how many times the product was purchased in that month.

Important: All requests made to the Backend Server must use the GET method of HTTP.

Should a malformed query be received by the server, a response code of 400 (Bad Request) will be sent and the request will be ignored.

In-Fridge Hardware to Backend Server Communication

As mentioned briefly above, the In-Fridge Hardware unit relies on HTTP to communicate to the Backend Server. The in-Fridge hardware sends two different HTTP messages types to the backend server. The first message is sent every 60 sec and it contains the temperature and humidity readings from the temperature/humidity sensor in a XML format.

To indicate the current temperature and humidity of the inside of the fridge, the In-Fridge Hardware unit would send a request to a URI of the form:

```
http://<server_ip>:<port>/TEMP_HUM
```

The HTTP request body then contains XML that defines the current temperature and humidity inside the fridge. An example of this XML is defined in Listing 3.

```
<temperature>10.1254</temperature>
<humidity>78.9034</humidity>
```

Listing 3: Example of XML Sent from In-Fridge Hardware to Backend Server

In the second message type The In-Fridge Hardware unit sends an XML representation of all of the items currently in the fridge. It is then the Backend Server's responsibility to determine which items were removed or added to the fridge.

To indicate the items currently in the fridge, the In-Fridge Hardware unit would send a request to a URI of the form:

```
http://<server_ip>:<port>/ADD_ITEMS
```

The HTTP request body then contains XML that defines the set of items currently inside of the fridge. An example of this XML is defined in Listing 4.

```
<itemset>
      <item>
            <id>0000110101</id>
            <upc>20202020202</upc>
            <expiry_date>20-08-2011<expiry_date>
      </item>
      <item>
            <id>00222222</id>
            <upc>333030303</upc>
            <expiry_date>20-08-2011<expiry_date>
      </item>
</itemset>
```

Listing 4: Example of XML Sent from In-Fridge Hardware to Backend Server

The In-Fridge Hardware does not expect any response from the server beyond the standard HTTP Response Header with code 200 (indicating everything is ok). Should the In-Fridge hardware send malformed XML, the server will response with a 400 error code (Bad Request).

Mobile App to Backend Server Communication

In order to present the items currently in the fridge, the iPhone application must receive this information from the Backend Server. This information is retrieved by an HTTP request with the following URI:

```
http://<server_ip>:<port>/SHOW?target=INFRIDGE
```

The HTTP Response Body then contains the XML definition of the InFridge ItemSet (as defined in the Object Hierarchy Section above). For a full definition of the XML format of objects, see the schemas within the "`FridgeSystem/schema/`" folder. After receiving the XML representation of the current fridge contents, the application parses this data and presents it to the user in an accessible format.

The iPhone application also sends information to the Backend Server when a user has manually tagged an item with a UPC or a leftover container. In either case, the iPhone application sends the information within the request body of an HTTP message to a URI such as:

```
http://<server_ip>:<port>/ADD_OBJECT
```

In the case of a manually-tagged item with a UPC, the request body contains the value of the UPC code. If a leftover item was added, the label supplied by the user is sent in the request body. The server then interprets the next item entering the fridge without a UPC code as having the value (UPC or name) as received from the iPhone application. If the message body does not specify a label or a UPC, the request is ignored by the server.

Web Server to Backend Server Communication

A standard web server (such as Apache) is utilized in order to provide the web-accessible content of the Smart Fridge System. The server *must* have PHP installed in order for the web content to be presented. Further, the Web Server and the Backed Server must reside on the same physical machine: the Backend Server produces HTML files that must reside within the Web Server's root web directory (as specified by the `DocumentRoot` property in Apache). In this way, the Web Server makes the files generated by the Backend Server web accessible.

HTML Assembly

The PHP running through the web server collects and presents the HTML generated by the Backend Server. The Backend server generated per-UPC HTML fragments that are amalgamated into a single webpage by a PHP script. For example, the file "`web/html/inFridge/011161133971.html`" contains on the HTML to represent all items with the UPC of 011161133971. All of the files within the "`web/html/inFridge/`" directory are assembled into a single HTML file, which then represents all UPCs currently in the fridge. In this way, whenever items are added to or removed from the fridge, only HTML *fragments* need to be generated – avoiding redundant generation of the HTML representations of all the files that *did not* change. Though this mechanism adds complexity to the PHP running on the web server, it significantly reduces the amount of XSLT processing required on the Backend Server.

Time-Dependent Content

It is sometimes necessary to regenerate certain HTML files. For example, the table that presents which items are expiry soon or the page presenting items that were removed recently must be regenerated in order for the transient notion of "soon" to remain valid. In order to trigger this HTML generation, the web server uses a PHP script that sends an HTTP request to a URI such as:

```
http://<server_ip>:<port>/GENERATE_HTML?target=TIME_SENSITIVE
```

This request is sent every time a user opens any of the following web pages:

- InFridge.php
- Expiry.php
- RecentlyRemoved.php

It is therefore ensured that all time-sensitive data present on the web site is up to date.

Plotting

The web server also presents plots to the user. The plot may show how many items were expired before being removed from the fridge (indicating potential waste) or how often a particular product was purchased within the past year. Though the plot is ultimately presented using JavaScript and CSS libraries, the data presented on the plot must be fetched from the Backend Server.

Data is received within the message body of a response to a URI such as:

```
http://<server_ip>:<port>/PLOT_DATA?target=EXPIRY_HISTORY
```

The x-values of the data represent months and the y-value represents the number of items that were expired when they were removed from the fridge of that month. These points are then passed into a PHP script that plots the data on a graph using Flotr libraries (Solutaire, 2011).

8 SOFTWARE SYSTEM

Auto-Generated Grocery List

The auto-generated grocery list feature uses an intelligent algorithm that monitors numerous parameters in order to automatically select the most appropriate items to be placed on a grocery list for the user. Recording of usage patterns gives the algorithm the necessary information to make informed decisions about which items should be added to the grocery list. The dynamic nature of a user's grocery is a problem we have considered in the design of this algorithm. For example, a user may buy an item one or two weeks in a row then lose interest no longer having a desire for that item. To achieve accurate results we must have some measure of desirability for each item. The central assumption that has been made is that an item which has not been purchased in a long time is associated with low desirability and should not be placed on the grocery list. The algorithm design incorporates a type of learning that yields more accurate results the longer you use the smart fridge system. The algorithm components will now be discussed in detail.

Algorithm Interface

The algorithm requires a number of input parameters in order to gain an understanding of the usage patterns in addition to the current state of the user's fridge. A series of item lists are used for this purpose. The Smart Fridge System keeps a record of what items are currently inside the fridge at any given time. This current item list reflects the present time state of the fridge contents. In addition to the current state of the fridge we need information about the history of items passing in and out of the fridge. This information is gained through the history item list which not only keeps track of what items passed into and out of the fridge but also the times in which these events occurred. The third and last parameter necessary is master item list. The master item list contains a single instance of every item which has even been recognized by the Smart Fridge System. The purpose of this list is to serve as a reference of all known items irrespective of being in the fridge currently or in the past. The master item list is also used to keep track of user specified parameters on an item per item basis. To summarize

three input parameters are used, the current item list, the history item list and the master item list. Upon completion of the algorithm an item list is produced containing the grocery list items.

Component 1

Certain items can be specified by the user to always be on their grocery list. Irrespective of all other factors an item flagged as "always on" will always appear on the grocery list if that item is not currently inside the fridge. Items labelled as "always on" are specified in the master item list by the user. The algorithm searches the master list for the "always on" flag set, then compares this item with the current item list. If the item is not found in the current item list it will immediately be added to the grocery list.

Component 2

This component attempts to predict how desirable a particular item is based upon the time since it was last purchased. The general assumption is items which have not been purchased recently are not likely to be of interest to the user any longer and should not be placed on the grocery list. The component functions in the following way. First, a list is generated that represents items which have high buying potential. This list is obtained by subtracting the items appearing in the current item list form the history item list. For the purposed of this component we assume the user wouldn't want any items that are still inside their fridge. The generated list is then searched. Each item in the list has an attribute containing the permanent exit time from the fridge (i.e. time of being entirely consumed). The exit time stamp is compared against a pre-determined threshold value. If the difference between the present time and exit time is below a predetermined threshold then the item is added to the grocery list. If the difference is greater than the threshold then we assume the user is no longer interested in this item and it is not added. A challenging factor was to determine a suitable threshold period. Too long of a period would mean that too many items are added to the list, many of which may not be of interest to the user any longer. On the other hand too short of period would result in items of interest not being added. Special consideration was given to find the most appropriate compromise. The threshold was decided to be 14 day. This decision was based on an average shopping period of once a week. If the user did not purchase the item within two grocery trips we assumed they were no longer interested in purchasing this item.

Component 3

One of the more challenging insights to have regarding the contents of your fridge is the approaching expiration dates. The Smart Fridge System is able to monitor each items expiration date and use this information to initiate a pre-emptive purchase. Based on our original assumption we can assume that items which are currently in the fridge are of high interest to the user. The algorithm searches through the current item list comparing each expiration date to the present date. If the difference between the current date and expiration date is below a pre-determined threshold the item is added to the grocery list. Again the challenge of find a suitable threshold value is seen. The threshold for this component was set at 7 days. The decision was based upon an average grocery shopping schedule of once per week. With a 7 day threshold the item is ensured to be purchased on the last grocery shop prior to expiration. Ideally we would like the item to be added only on the last grocery shop before expiring. If the threshold were set too high then the pre-emptive purchase would occur too soon. The result could be wasting of

food as the expiration date of the pre-emptively bought item would be earlier than if we had waited until the last moment.

Component 4

The fourth component is used to predict when an item is nearing the end of its contents. This portion of the algorithm is adaptive and learns, the Smart Fridge System yields ever improving results as time progresses. The Smart Fridge System assigns to each item an entry time and an exit time. The entry time is the time for which the item was first placed in the fridge while the exit time is when the item was permanently removed from the fridge. By taking the difference between exit and entry times we can gain insight to the consumption time of this item. As the user goes through repeated cycles of buying a particular item and consuming it, we can average over all these individual consumption times. As the number of cycles increase, the average consumption time becomes a better estimate of the actually consumption time. In addition to the average consumption time, a current consumption time must be calculated. For every item in the fridge, a current consumption time is calculated by taking the difference between present time and entry time. This number represents the amount of time the user has been consuming the item. To complete the decision of whether or not to add the item again a threshold factor is needed. The threshold value for this component is a percentage of total consumption. For example, we might have calculated the current consumption time is 90% of the average total consumption time for this item. We would then choose to add the item because there is a good chance that it is nearly entirely consumed. As with prior threshold values a compromise is seen. If the threshold is set too low then items would be added to the grocery list too early and might result in wasting of food. Similar to the prior situation with expiration dates. The user may not be able to consume the rest of the previous item and the new item before the new items expiration date. On the other hand if the threshold is set too high then the item may be consumed before the user has a reasonable chance to purchase the item during their next trip to the grocery store. The decision was made to set this threshold to 90%.

Component 5

The final component of the algorithm allows another user specified value to control what is added to the grocery list. Some users may want the option to specify a quantity of an item they would always like to have in the fridge. For example, the user may say I would like to have 3 bottles of apple juice in the fridge at all times. The algorithm will search the current item list and make sure this requirement is satisfied. If the quantity of an item is less than the user specified quantity then it gets added to the grocery list. The difference between the user specified quantity and the current quantity is the value added to the grocery list. If the item cannot be found in the current item list it then it gets added to the grocery list with the user specified quantity.

Summary

The smart fridge auto-generated grocery list feature uses an intelligent multivariable algorithm illustrated below. A number of important factors are considered using recorded usage patterns in addition to user specified information. We have focused our attention to five main components we believe to yield the most accurate grocery list possible. The first component gives the user control to

make sure a particular item is never forgotten irrespective of any other factors. The second component predicts the desirability of an item based on history usage patterns. The third component pre-emptively identifies items which have expiration dates in the near future. The fourth component identifies items which have a high probability of being consumed entirely in the near future. The fifth and final component allows the user to ensure a specified quantity of an item is always available in the fridge. The culmination of all five components gives a comprehensive image of what items the user is most likely to have on their grocery list if they were to manually write it themselves.

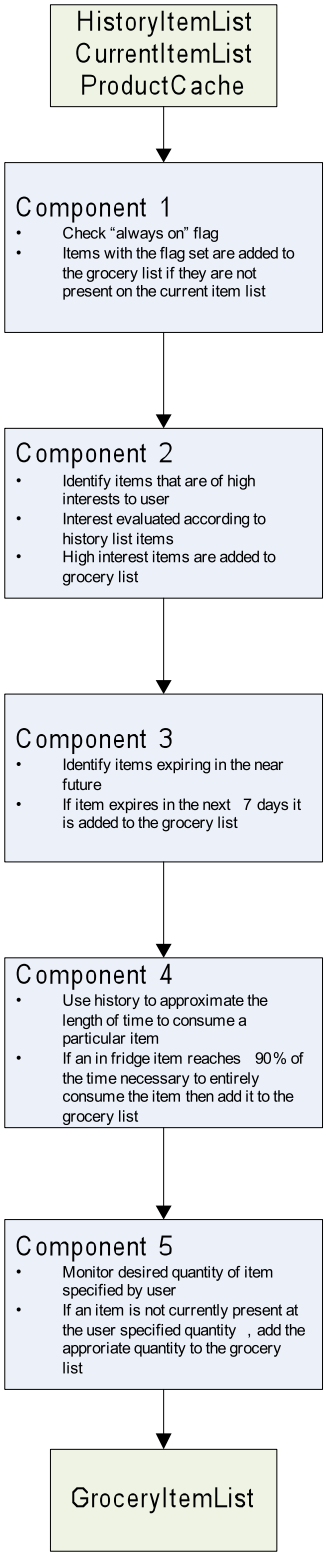

Figure 10: Auto-generated Grocery List Algorithm

Nutrition Metric System

Identifying the nutritional value of a food item is a difficult task for most of the population. The Smart Fridge System incorporates a nutritional metrics system used to present nutritional information in a concise, easy to understand format. The nutritional value of an item is challenging to present because nutritional value is represented by numerous parameters. For example, the amount of fat, cholesterol, carbohydrates, protein, vitamins etc. all must be considered to gain perspective about how nutritious an item is. The Smart Fridge System uses a mathematical model to reduce this multivariable problem to a single number that represents the nutritional value. The model is described below.

Nutritional Profiling Model

The model we used in the Smart Fridge System was based upon a concept called nutritional profiling. The idea is to consider the main components of a food, namely fats, protein, carbohydrates etc. and quantify the entire nutritional value with a single number. We chose to use a model discussed in a paper called "Modeling Expert Opinions on food Healthiness: A Nutritional Metric." In this paper they surveyed the leading U.S nutrition experts about the healthiness of sample food and beverage products. Each expert was given a number of sample foods together with corresponding nutritional information. After given time to analyze the sample, the expert was requested to provide a score between -5 to 5. -5 represented extremely unhealthy while 5 represented extremely health. After all the experts had made judgements on all the items a linear regression was used develop an equation for the results. The equation considers the 12 parameters provided by a standard nutritional facts label. Using this model we were able to quantify the nutritional value of a single item.

System Description

The nutrition metric system relies on two classes for its functionality. The Nutrition Item class is used to represent the nutritional data of an item while the Nutrition Metrics class operates on the Nutrition Item object. The two classes are described in further detail below.

Nutrition Item Class

The nutrition item is a class used to represent the nutritionally relevant information about a particular item. Attributes include all information present on a standard nutritional facts label in units of percent daily value. The nutrition profiling model is implemented in this class as a method. A few other methods were necessary to convert percent daily values to units of grams. The unit conversion was necessary for the use of our nutritional profiling model.

Nutrition Metrics Class

The nutrition metrics class is composed of a number of functions that use the nutrition item class to provide the core functionality of the system.

Construct Nutrition Item

The item lists used to record the current and history of the Smart Fridge System discussed in the previous section do no store nutritional information about those items. This decision was made to conserve space by preventing the duplicate storage of nutritional data about each item. The ProductCache list is used to store a single instance of an item together with its nutritional information

stored inside a hash table. Nutritional information is recalled from this hash table using key value pairs. The key is the name of the nutrient while the value is the percent daily value of that nutrient. The getNurtritionItem function provides the construction of a nutrition item object given an item from the current item or history item lists. The function identifies the item in the product cache and gets nutrient information via the hash table. The nutrient item is then return from the function.

Top Five Lists

The top five lists function is used to identify the 5 most and 5 least nutritious items in a list. It uses a single input parameter, a list of items. This could be used for the current item list to construct a top five list of items currently inside the fridge. In addition, it could be used on the history item list to see the most healthy and least healthy items ever consumed. The function operates by calculating a nutritional value for each item in the input list and storing the value in an array. The array is then searched for a maximum and minimum nutritional value, recoding the index in the array. The item with this index is then added to the top five list array. The item just added are then removed and the process is repeated to find the next maximum and minimum values. This process is carried on until the top 5 most and least nutritious items are identified and stored in the output array.

Average Nutritional Value of List

This function is used to calculate the average nutritional value of an entire item list. The function takes a single input parameter, an item list, and returns the average nutritional value of the list. The calculation is accomplished in the following way. For every item in the list construct a nutrition item object. Next, call the method to calculate the nutritional value for each on the newly created nutrition item objects. Take the sum of all nutritional values and divided by the number of items in the list to get the average nutritional value. The average value is then returned.

Items in Range

This function was necessary as a helper to accomplish other calculations in the system. The input parameters are a list of items and two dates. The two dates represent a start date and an end date. The function searches through the item list for items which have an exit fridge time that occurs in between the start and end dates. The function returns a list of items that have exit times that fall within the specified time span. Since the exit time represents the time in which an item permanently left the fridge this function can be used to identify what items were entirely consumed within a time span.

Nutritional Value versus Time

The final function is used to generate a plot of nutrition value of consumed items over time. The input parameters are a list of items, normally the history list, and an integer representing the number of weeks into the past to be analyzed. The output is given an array, each index stores the average consumed item nutritional value for a single week. We decided to use a single week resolution, because this ensures a high probability of at least a few items being consumed within that time period. If the resolution was to be a single day then days where no food was consumed would have an average nutritional value of zero. The calculation is accomplished in the following way. The specified time period is broken up into single weeks into the past. For each week, a search is performed to identify all items which were consumed within that week. The "items in range" function described above was used for

this task. Once a list of items consumed for the week was constructed we calculated the average nutritional value of that list. A normalization factor was used to account for how long it took to consume an item. For example, an unhealthy item may have been consumed only very slowly and should therefore hold less weight in the averaging calculation. In contrast an item consumed very quickly should hold a higher weight in the averaging calculation. Once the average nutritional value is calculated it gets added to the output array. The process is completed for every week specified by the input parameter week value. Upon completion the array is returned. Each array index holds the average nutritional value for all items consumed in that given week.

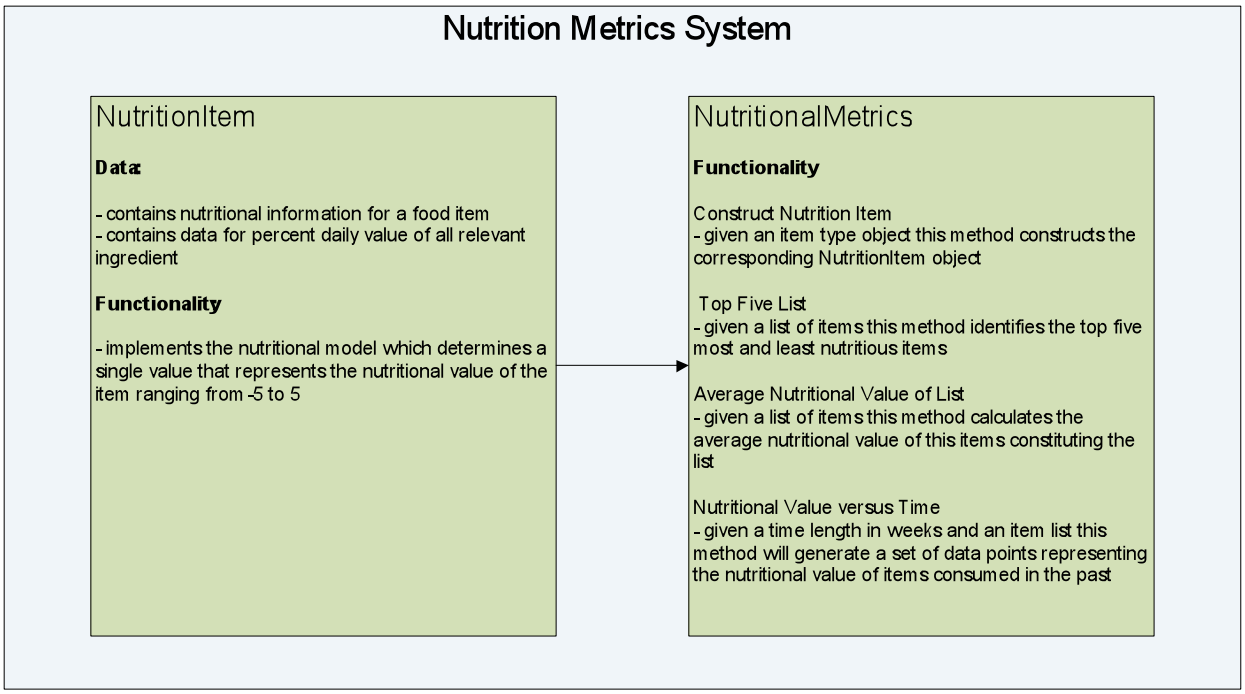

Figure 11: Nutritional Metrics System Structure

9 MOBILE APPLICATION

Overview

The mobile app was designed using the Objective C programming language to run exclusively on the iPhone 3GS and the iPhone 4. The purpose of the mobile app is to provide a clear, useful, easy to use user interface for the user to access information from their RFID Smart Fridge. The five main components of the mobile app are "In my Fridge," "Expiring," "Grocery List," "Eat Healthy," and "Scan Barcode" as denoted by the five icons along the bottom of the screen in the figure below. The user can switch between these displays easily by simply clicking or tapping the screen over the desired icon. "In my Fridge" displays an inventory of the items currently in the fridge organized by category, as seen in the figure below. When the user clicks on "Expiring" a similar display showing all the items currently in the fridge and their expiry date, organized from the items which will expire soonest to latest. Clicking the grocery list will display a grocery list predicted for the individual user based on their consumption

history (see the Auto-Generated Grocery List Section under Software System). The "Eat Healthy" display will display the fridge's current temperature and humidity at the top of the screen. This will be followed by a list of the top five most nutritious and top five least nutritious items in the fridge (see Top Five Lists section under Nutrition Metrics). From there the user can scroll down to see a graph of the average healthiness of all the items consumed from their fridge versus time (see Nutritional Value versus Time section under Nutrition Metrics). Finally, the "Scan Barcode" display will show a button called Scan. Once Scan is clicked the app will bring up the camera and a message which asks the user to "adjust the camera so the barcode of the item to be added is in view then tap the screen". Once the user taps and the barcode is verified a message to "please add item with reusable RFID tag to fridge" as well as a cancel button will be displayed until the user adds the item to the fridge of clicks cancel.

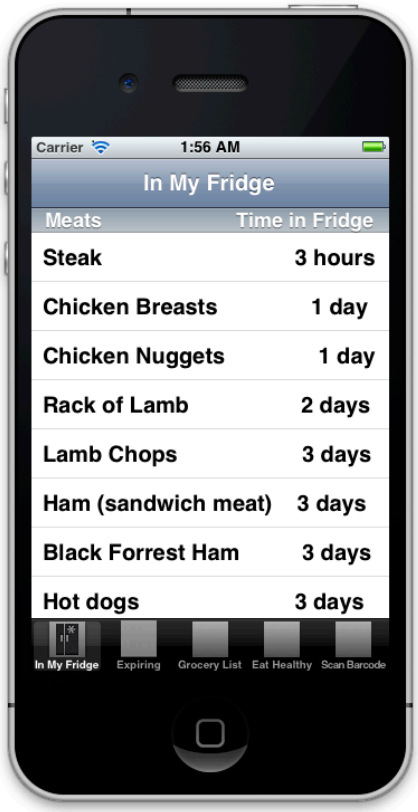

Figure 12: Mobile App-In My Fridge Display Screen

Mobile App System Design

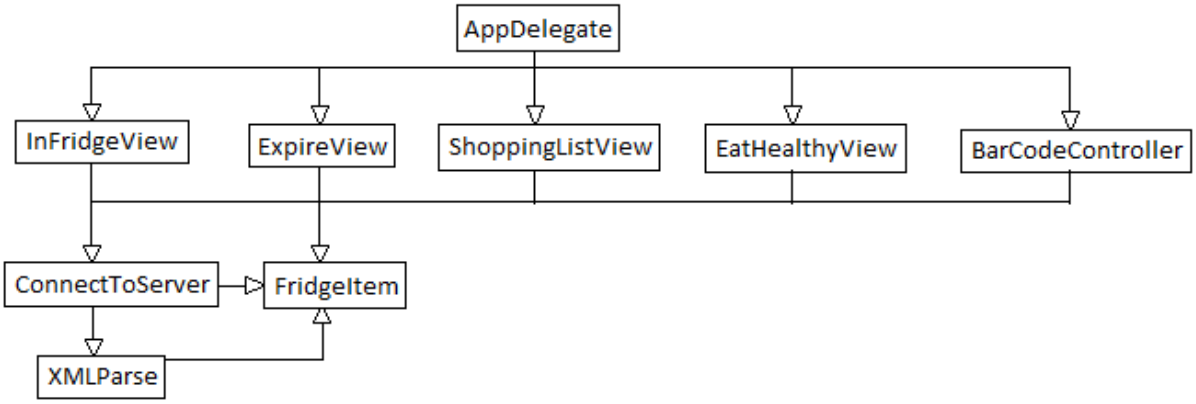

Figure 13: Mobile App Class Hierarchy

The application is initialized at the top by the AppDelegate class which instantiates the object tabBarController of type UITabBarController. This object controls the tab bar seen at the bottom of the screen in Figure 12 (above), as well as each view (also objects) associated with the tabs. The AppDelegate then proceeds to instantiate five view objects from each of the classes InFridgeView, ExpireView, ShoppingListView, EatHealthyView, and BarCodeController. The AppDelegate then associates these views tabBarController, such that these five classes respectively control the views of each of the five tabs seen at the bottom of Figure 12 from left to right.

The four view controller objects created from the classes InFridgeView, ExpireView, ShoppingListView, and EatHealthyView each display different data (describe in the Overview section above) drawn from the backend server. They each create an object, connectToServer, using the class ConnectToServer which, when that particular view is being displayed, sends a request (see Mobile App to Backend Server Communication section) every thirty seconds checking for updated data. The backend server then returns an xml file containing the requested list of items. For instance the "In My Fridge" view would send a request to the server and would receive an xml file containing information on each of the items currently in the fridge. These xml files are then parsed by each view controller object using the class XMLParse and the data for each item is stored in objects created from the class FridgeItem. The view controller objects then display this data in a table created from the class UITableView. Each of the four view controller objects customizes their table.

InFridgeView customizes the table it displays to show each item currently in the fridge and how long it has been in the fridge, as well as items that have been recently removed from the fridge (within the past twenty minutes) and how long ago they were removed. These items are arranged according to category as described in their UPC code. ExpireView shows each item in the fridge and their expiry dates arranged from the item that will expire in the shortest period to the item that will expire in the longest

period of time. ShoppingListView will display an auto-generated grocery list (see the Auto-Generated Grocery List section under Software System). Finally EatHealthyView will have a table containing the top five healthiest and least healthy foods as well as the fridge's current temperature, humidity, and a graph of the average healthiness of the items in the fridge vs time. This graph will be generated within the EatHealthyView class.

The BarCodeController class creates the view describe in the Overview section (above) where the user presses the "scan" button to scan a bar code of an item without an RFID tag using the iPhone's camera. The BarCodeController class then uses the ZBarSDK library to analyze the image of the barcode and determine its associated UPC code, which is a numerical value. This UPC is then sent to the backend server (see Mobile App to Backend Server section under Communication). The backend server then associates this item's UPC code with the next reusable RFID tag that it detects being added to the fridge and adds this item to its inventory.

10 TESTING

Simulated Fridge System

In order to rigorously test the Smart Fridge System we needed a way to generate a year of usage data. This task was difficult for numerous reasons. The primary challenge was to develop a method for generating an item history that would accurately represent an average person's usage patterns of the Smart Fridge System. In the ideal situation we would have monitored several people's fridges over the course of a year and taken note of every item passing in and out of the fridge along with expiration dates of each item. Unfortunately this long research processes would not be practical for the four month duration of our project. The alternative was to develop a method to generate a yearlong history of items without explicitly specifying every parameter manually.

Generating a year's worth of grocery items by hand would be a daunting and tedious task. First, you must select every single item that would appear in the history. The item must be chosen appropriately based upon what else in currently in the fridge. For example, it doesn't make sense to have a fridge full of just meat or condiments. At each instance in time the fridge should contain a reasonably realistic balance of items. Next, the entry and exit times of the item must be explicitly specified and carefully chosen to represent a reasonable amount of time spent inside the fridge. The expiration date must also be specified and chosen appropriately based upon when the item entered the fridge. One of the most difficult challenges it to have a balance of items temporally as items are constantly being added and consumed. This necessity of balance is essential to testing the smart fridge nutritional metrics system. Without a balanced set of items there would be no way of knowing how well the nutritional metrics system performs as the results would appear erratic. In the case of a fridge full of condiments the nutritional metric would classify the fridge as extremely unhealthy. Yet this situation is obviously not useful as mostly people don't have a fridge full of condiments. Clearly, hand crafting the history item list would be a difficult task. Rather than explicitly setting all the necessary parameters we decided to make

some generalized requirements that must be satisfied and then used random number generation to get reasonable values.

Achieving a realistic set of items without manually specifying each value required a significant amount of thought and design. We need a way to set some general guide lines for what was expected out of the history list and then use random number generation to explicitly set the values. Before we could attempt to find a solution we needed to identify what made a history item list realistic. We first began to research what a person should be consuming if they had a relatively balanced diet. During the research phase, on numerous occasions, the sources specified a diet on a week by week basis. The decision was then made to hand construct a few sets of data that would constitute a week's worth of grocery items. This way we could ensure that every week contained a well-balanced set of items. The further assumption was then made that groceries would be bought on a weekly basis. Once the weekly item lists were constructed a random number generator was used to randomly select a week's worth of groceries to be added to the history list. Once all the items were put into the history list the next task was to specify the entry and exit times and expiration dates.

Entry dates was a drastically simplified by the design since we agreed that a week's worth of groceries would be bought weekly. Therefore, every consecutive week's worth of items had an entry date 7 days apart from the previous week's worth of items. The next parameter to specify was the exit time of the item. The time of exit was more difficult to specify than the entry time since this would be different for every item. Intuition would tell us that if we were buying groceries on a weekly basis then the items would be consumed on a weekly basis also. It would be reasonable to assume the consumption of the items would follow a Gaussian distribution. Items would be most likely to be consumed part way through the week, with less items being consumed much earlier or later. If we set the mean of the Gaussian distribution to 5 days after the entry time we would then see some items would be consumed before and after the 5^{th} day. It follows that a smaller percentage of items would be consumed very soon after the entry time and very long after the 5^{th} day. This agrees with our choice to model this behavior with a Gaussian distribution. In order to determine a reasonable standard deviation for the Gaussian distribution we assumed that most of the items should be consumed by the 11^{th} day after the entry time. With a standard deviation of 3 we would ensure that 95% of the items are consumed within the entry time and 11th day from the entry time. Again this choice agreed well with the situation we were trying to model. The third and final parameter we needed to specify was the expiration date. The expiration date was handled in a similar manner to the exit date. The expiration date assumption was a little more unrealistic than the exit time since the expiration data is heavily dependent on the type of item. For example, frozen foods would last much longer than unfrozen vegetables. For our purposes we are prepared to accept this particular short coming of the model. The single requirement we must satisfy is having some items expire prior to exiting the fridge. We needed this particular situation in order to test algorithms used to detect expired items in the fridge. To satisfy this requirement we needed to determine the appropriate mean and standard deviation of the Gaussian distribution. The mean and standard deviation of the Gaussian distribution for the expiration dates should have some overlap with the Gaussian distribution used for the exit dates. We wanted the upper 2.5 percent of the

exit dates to fall within the lowest 2.5 percent of the expiration dates. To achieve this, the mean was chosen to be 21 days from entry time with a standard deviation of 3.

Through research and understanding of the characteristics in a balanced grocery list and human behaviour we were able to avoid a tedious task with a more elegant solution. Recognizing how to model the randomness of the situation together with guidelines and restrictions we were able to accurately simulate a yearlong history of grocery items for an individual.

Using this model we will be able to develop a number of different random consumption histories which we will then artificially insert into the server for testing purposes. These artificial histories will be used to test the mobile app and the website. We plan to test each of the features of both the website and the mobile app (see Web Presentation and Mobile App sections in our Requirements Document or Mobile App and Web Server to Backend Server Communications sections in this document) using at least twenty different artificial histories in order to make sure that all of the desired information in these example histories is communicated accurately and effectively to the user.

A further series of tests will be conducted using the entire RFID Smart Fridge system where RFID tagged items will be placed in and removed from the fridge by our team members over the course of several days with the entire system up and running. A written log will be kept of all items added and removed from the fridge. At the end of the test period this data will be compared with the data found in the server and the data displayed by the mobile app and the website to insure that the system is working as specified.

In addition, a series of unexpected but possible test cases (listed below), such as malformed request being sent from the mobile app to the backend server, will be tested in order to insure that the system will not crash under any circumstances.

Finally, a usability test will be developed based on the final RFID Smart Fridge product and test subjects unfamiliar with the RFID Smart Fridge will be asked to use the product and give feedback. This will insure that our engineers don't overlook any usability issues because they are too familiar with the system. We will analyze this user feedback and use it to insure that all aspects of the RFID Smart Fridge are easy and convenient to use and understand.

Communication Testing

Integration testing was performed through verifying processes that involve cross-component communication. All tests outlined below:

Backend Server Testing

- All paths were tested (Table 3)
    - Server-side behavior verified
    - Response behavior verified
- Non-existent paths were tested
    - Error response behavior verified

In-Fridge Hardware to Backend Server Communication Testing

- Sending sets of items, verifying response (HTTP code 200 – OK) and server action
    - Test handling of sets of various sizes:
        - Empty set
        - Single item set
        - Multiple item set (10)
- Sending HTTP requests to an invalid path, verifying resulting error response

iPhone Application to Backend Server Communication Testing

- iPhone sends a SHOW request to the server, parses response
- iPhone sends malformed request to server, handles error
- iPhone sends a leftover label through a ADD_OBJECT request, receives code 200 (OK) response
- iPhone sends a UPC code through a ADD_OBJECT request, receives code 200 (OK) response
- iPhone sends malformed ADD_OBJECT request, receives 400 (Bad Request) response

Web Server to Backend Server Communication Testing

- Verify time-sensitive web pages are updated (dates change according to the current date)
- Verify plot data as seen on the web page is correctly populated
- Ensure web pages update after HTML is regenerated (updated) as a result of Backend Server events

11 CONCLUSION

This document describes in detailed the proposed design for the RFID Smart Fridge. This design will be adhered to as closely as possible during the development of the RFID Smart Fridge prototype system. This will insure that all parts of the design are coherent and that no aspect is overlooked. The test plan will insure that all desired functionality of the RFID Smart Fridge is achieved and that no bugs exist in the software. This design provides clear goals for the development of the RFID system and prototype and will allow us to complete the system on schedule by December 7, 2011.

12 REFERENCES

[1] R Pagliara, D Jungic, M Joblin, and S Verner, "RFID Smart Fridge Functional Specifications," ENSC 305, 2011.

[2] AIM Global. (2010) RFID Glossary. [Online] *[last accessed Nov. 20th, 2011]*. http://www.aimglobal.org/technologies/rfid/rfid_Glossary.asp

[3] GS1. (2011) EAN UPC barcode. [Online] *[last accessed Nov. 20th, 2011]*. http://www.gs1us.org/standards/barcodes/ean_upc

[4] technovelgy.com. (2010) How RFID works. [Online] *[last accessed Nov. 20th, 2011]*. http://www.technovelgy.com/ct/Technology-Article.asp?ArtNum=2

[5] US Food and Drug Administration. (2010, Jan) FDA study determines RFID effects on implanted cardiac devices. [Online] *[last accessed Nov. 20th, 2011]*. http://www.rfidnews.org/2010/01/07/u-s-food-and-drug-conduct-study-to-determine-rfid-effects-on-pacemakers-and-implanted-cardiac-devices

[6] RFID 4SME and the European Union, "Is RFID safe at the workplace?," European RFID Platform, Brussels, 2008.

[7] M Ward and R van Kranenburg, "RFID: Frequency, standards, adoption and innovation," JISC Technology Standards Watch, 2006.

[8] GlassFish. (2011) JAXB Reference Implementation. [Online] *[last accessed Nov. 20th, 2011]*. jaxb.java.net

[9] SimpleUPC. (2010) SimpleUPC-Product Lookup. Made Simple. [Online] *[last accessed Nov. 20th, 2011]*. simpleupc.com

[10] R Fielding, J Gettys, and J Mogul. (1999) Hypertext Transfer Protocol -- HTTP/1.1. [Online] *[last accessed Nov. 20th, 2011]*. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[11] Solutaire. (2011) Flotr Libraries. [Online] *[last accessed Nov. 20th, 2011]*. http://solutoire.com/flotr/

[12] Gson. (2011) Gson RoadMap. [Online] *[last accessed Nov. 20th, 2011]*. https://sites.google.com/site/gson/gson-roadmap