Debreceni Egyetem
Informatikai Kar

RFID TECHNIQUES AND IMPLEMENTATION
Design and development of gate-based RFID systems for retail and libraries

Témavezető
Dr. Juhász István
Egyetemi adjunktus

Készítette
Agócs László Péter
Programtervező matematikus

Debrecen
2006

Abstract

The aim of the work is to give an introduction to recent developments and standards in radio frequency identification technology and to provide an overview as well as implementation details for the components in the RFID architecture of a retailer or a library branch. While mostly concentrating on the latest – EPCglobal's Class1 Gen2 - ultra-high frequency protocol, high and low frequency technologies and their core properties are also discussed.

The process of developing and implementing an actual system is presented through the description of a retail self-checkout system made for demonstration purposes in the laboratory of the University of Applied Sciences as a sub-project of the University's LASSO project. The core hardware component was a self-built RFID gate. All the software including the modules for communicating with the RFID reader, a simple middleware-like server, and applications for the end-user and for internal testing were self-made. The software components were written in C# and are thus targetting the .NET Framework 2.0. Besides describing the low-level hardware and software issues of this proof of concept architecture, an outlook is given to the most common concepts found in today's widely available equipment, middleware systems, and standards. Some areas of software development, like multithreading and network programming, that are critical for RFID-enabled applications are discussed in detail.

Kivonat

A munka bemutatja a rádiófrekvenciás azonosítás (RFID) technológiák napjainkban elterjedt, széles körben elérhető változatait, különös tekintettel a kiskereskedelmi és könyvtári környezetek RFID rendszereinek komponenseire. Az ilyen rendszerek vizsgálatára – hardver és szoftver szempontból egyaránt – a finnországi Jyväskylä University of Applied Sciencies számára készített demonstrációs rendszer bemutatásán keresztül kerül sor. Ezen projekt célja egy automatikus, személyzet nélküli fizetést (a vásárló kosarában elhelyezett termékek azonnali, észrevétlen azonosítását) lehetővé tevő RFID alapú kapu és az azt működtető szoftverrendszer kifejlesztése, illetve a rendelkezésre álló eszközökkel történő megvalósítás korlátainak vizsgálata volt. A rendszer elsődlegesen UHF alapú eszközökre épült. A projekt során megtervezett és megvalósított szoftverrendszer teljesen saját fejlesztésű, beleértve a hardvereszközök vezérlését és a végfelhasználó számára a prezentációs felületet biztosító komponenseken kívűl a fizikai eszközök teszteléséhez, összeállításához szükséges segédalkalmazásokat is. Ezek ismertetése során a kisebb méretű RFID alapú rendszerek fejlesztésekor fokozottan jelenlevő programozási területek (többszálúság, hálózati kommunikáció) egyes problémái is tárgyalásra kerülnek. Mindemellett nem hiányozhat néhány fontos szabvány és specifikáció főbb koncepcióinak ismertetése, valamit a napjainkban oly divatos biztonsági kérdésekre történő rövid kitekintés sem.

CONTENTS

1 OBJECTIVES....3
2 INTRODUCTION TO RADIO FREQUENCY IDENTIFICATION....4
3 PHYSICAL COMPONENTS OF THE RFID SYSTEM....5
3.1 Tags and readers....6
3.2 Operation in the ultra-high frequency spectrum....9
3.2.1 EPC Class-1 Generation-2....10
3.2.2 Limits and regulations....11
3.3 LF and HF RFID technologies....13
3.4 Device interfaces....14
3.5 Privacy considerations....16
4 SOFTWARE COMPONENTS....18
4.1 Reader protocols....18
4.1.1 The Reader Protocol Standard....19
4.1.2 CHUMP....21
4.2 Middleware....22
5 IMPLEMENTATION OF THE DEMONSTRATION SYSTEM....23
5.1 Architecture....24
5.2 Use cases....25
5.3 Hardware components....27
5.4 Building the system....29
5.4.1 Setting up the readers....29
5.4.2 The gate....31
5.4.3 Testing....35
5.5 Common programming issues in communication....36
5.5.1 Multithreading....36
5.5.2 Network programming....46
5.6 The server application....49
5.7 The end-user application....57
6 RFID NOW AND IN THE NEAR FUTURE....61
SUMMARY....63
REFERENCES....64

APPENDICES..... 66
Appendix 1: Presentation of the system at Tekniikka 2006..... 66
Appendix 2: Plans of a simple RFID library system..... 69
Appendix 3: List of abbreviations.....71

1 OBJECTIVES

The main target audience of the work are the engineers, developers, project managers, and other workers taking part in small- and medium-scale RFID-related projects. The imaginary starting situation is the following: an RFID kit containing some strange looking device (the reader), some antennas, cables, some sample tags, and a CD with some software and often scarce documentation arrives at the premises of the company. There are preliminary plans, visions, and ideas about how the end result should work and look like but nobody knows for sure how to configure and program the equipment, and there may be doubts if the device and its accessories are suitable for the given tasks. Naturally the audience who might greatly benefit from reading through the following sections is much wider, the above situation is emphasized because the development of our shopping demonstration system also started like that.

Besides describing the core hardware and software concepts of today's RFID systems, the design and implementation of a specific project is discussed in details, covering both hardware and software components, mostly from a programmer's viewpoint. Some software development concepts, which are increasingly important when writing RFID-related applications, are also given a closer look.

The project implemented at the Jyväskylä University of Applied Sciences demonstrates one of the biggest promises of RFID: the shopping experience of the future where customers are not held up by queues and cashier desks, people can just walk around carelessly and the system will “magically” know what is inside their shopping cart.

2 INTRODUCTION TO RADIO FREQUENCY IDENTIFICATION

RFID stands for Radio Frequency Identification. It is one of the most often heard terms today and one of the technologies with the biggest promises. Its core function can be described simply as identification of all kinds of physical objects. This identification is done without any physical contact. Of course there are proven technologies, like bar code reading, that offer the same without touching the items, but they usually require a clear line of sight since they use optical methods to collect some kind of information. With RFID this is not required. Items to be identified can be hidden in one's pocket, bags, inside a box, etc. The “item” can be a living creature, animals or people wearing some little device or having it embedded into clothes, or even having it implanted into the living tissue. Regardless of the lack of any physical or optical contact, RIFD still works, thanks to radio waves, and humans are not able to sense that something is happening, even when their items or they themselves are just being identified. In the end, when they realize that something really happened, it might seem like some kind of magic to them.

What is the use of identifying objects from the real world? Just like with bar code reading at the local supermarket, where the cashier uses a device to transfer some data - encoded and printed onto the packaging of goods in a special way - into the computer system, the goal is to recognize the items in the system's surroundings. Humans can do this with the help of their eyes and their brain but computer systems need some additional help. When this problem is solved, i.e. the items have been “marked” in the appropriate way (have a bar code printed on them or have some device attached), the system will be able to operate accordingly to the items it “sees”, without human interaction.

In case of bar code reading at a cash desk or during the check-out of books at library the goal is simple: transferring information into the computer system without wasting excessive human power and getting it processed there without any human interaction. But the number of possibilities is huge, or even endless, the only limit is our own

creativity, especially since RFID is not limited by the traditional problems of identification technologies.

Automatic recognition of piles of books during check-out and check-in at libraries is now reality. Tolling systems using radio frequency at motorways have already been installed throughout the world. RFID is widely used in identifying and tracking animals without even going close to them. An intelligent shelf or desk may know what items it holds and an attached computer may search for and provide related information. There are the visions about intelligent refrigerators that “know” what is inside them and warn if a product has expired or had been taken out permanently (meaning that it has been consumed and more should be ordered from the local grocery store). RFID-based systems for tracking and identifying luggage or even people are in test phases at some airports. In future supermarkets customers will not be held up by long queues and slow bar code reading, with RFID the goods in one's basket can be identified by simply walking through a gate. A huge driving force in RFID development was the logistics and supply chain environment. One can think of large boxes and containers just arrived from a factory, moving on a conveyor belt. Instead of relying on people standing near the belt and watching the incoming boxes, the computer system can identify them without any human help, route the boxes to the proper destination, and keep an up-to-date track record of all the incoming items, and all of this can be done at much faster speed.

3 PHYSICAL COMPONENTS OF THE RFID SYSTEM

The two core and most visible components of an RFID system are the readers and the tags. Sometimes the term “interrogator” is used in place of “reader”. The basic model of operation is the following: The reader transmits an electromagnetic signal at a given frequency repeatedly with short idle intervals. The tags that are in range pick up the transmission and respond by transmitting the signal back, information is sent by modulating the signal.

To be more precise, it must be noted that of course it is the antenna, in both cases, that transmits and receives the signal, however, in case of tags it is often unnoticeable and sometimes even hard to believe at a first look that there is an actual antenna in that small label, button, capsule, or plastic card. Similarly, many readers, especially those that are designed for proximity reading, come with integrated antenna which may be well hidden, just like in case of modern mobile phones.

A tag protocol describes how the data is sent and received from the tags. The properties and operation depends heavily on the set of frequencies used for communication. Most RFID systems today use a small set of frequencies from the low-frequency (LF, 30-300 KHz) band, the high-frequency (HF, 3-30 MHz) band or from the ultra-high frequency (UHF, 300-3000 MHz) band.

3.1 Tags and readers

RFID tags usually consist of a microchip, memory, and the antenna. They come in many forms and shapes, like smart labels (the antenna and circuits reside in layers of paper), plastic cards, coins, buttons, disks made from plastic or PVC, capsules that can operate also when put in liquid material, small tags embedded into clothing, other products, or even into the human body (e.g. under the skin).

Passive tags do not have any power source and so they are not able to communicate (emit any signals) unless they are energized by the transmission of the reader. In contrast, active tags, that have a power source (usually a battery) may even be able to communicate with each other without the presence of a reader. A battery may be responsible not only for the communication, but it may power a processor, memory, sensors, and other circuits integrated into the tag.

Most tags store at least a value that consists of 40, 64, 96, or more bits. The purpose of this number is to uniquely identify the object to which that tag is attached. This identifier will be referred to as “ID” or “tag ID”. Besides this, tags may store other information, like CRC values, passwords, and user data, depending on the tag type and

protocol. If the tag is read only then the ID is fixed during manufacturing. With write-once or rewritable tags the user is able to change (program) the value one or more times. It is nowadays not hard to find rewritable tags that can store a hundred or more bytes of used-defined data. It might be tempting to use this space for storing various product details about the item to which the tag is attached but excess care must be taken because of the possible privacy issues (see Section 3.5). The capacity of tags is increasing constantly while the physical size is getting smaller and smaller.

Figure 1. About a hundred EPC Class 1 Generation 2 sticker tags, just as they left the factory.

The smart label kind of tags can be applied to items and product automatically using dedicated equipment, or manually, tags suitable for manual attachment usually come in sticker form. (see *Kleist, R. et al 2005.* about RFID labeling)

Figure 2. An ISO 15693 HF tag that is indistinguishable from a regular plastic card.

Tags having a card or button format are ideal for identification and access control purposes.

Figure 3. A tag for low-frequency RFID.

A reader uses its antenna to send digital information encoded in a modulated waveform. Some readers have an integrated antenna while others are designed to have one or more external antennas attached. In many cases the term “reader” can be misleading as such a device may also be able to write tags. While a reader is designed for operation at a specific set of frequencies (LF, HF, UHF), the advanced ones often have multiprotocol capabilities which means that they are able to read different types of tags (e.g. an UHF reader may support ISO 18000-6 A/B, EPC Class 0, Class 1, Class 1 Gen 2, etc., of course some features may not be supported with all types of tags).

Figure 4. The SAMSys MP9320 UHF reader.

Two important characteristics of antennas are the pattern and the polarization. The former defines the reading area and the latter is the orientation of the transmitted electromagnetic field. With linear polarization the reading distance is typically longer but the coverage area is smaller and provides less tolerance for random tag orientations. Circular polarization means that the energy is radiated in a circular pattern from the antenna, the coverage area is higher, but the maximum distance is reduced. For UHF applications typically externally connected patch antennas are used.

3.2 Operation in the ultra-high frequency spectrum

It is the 860-960 MHz band that is usually used for this type of RFID communication but in most countries only a part of this spectrum is available freely, i.e. without a government license. See Section 3.2.2 for more discussion on such regulations. Out of the LF-HF-UHF trio it is the UHF technology that provides the highest performance regarding speed and reading distance, and is the most suitable for highly populated environments, i.e. where many tags can appear at the same time. With passive tags the maximum reading distance can be as high as 9 meters in theory. In practice usually a few meters are achievable.

The following passive UHF tag classes and specifications exist and are in use at the moment:

Class 0	Read only, 902-928 MHz
Class 0+	Rewritable, 902-928 MHz
Class 1	Write once, 902-928 MHz
Class 1 Generation 2	Rewritable, 860-960 MHz
ISO 18000-6A	Read only, 862-870 MHz
ISO 18000-6B	Rewritable, 860-930 MHz
ISO 18000-6C	Class 1 Gen 2 was approved by ISO as 18000-6C during 2006

Table 1. UHF tag classes.

Besides Class 0 and 1, EPCglobal, Inc., a successor of the Auto-ID Center, defines the following classes for future tags:

Class 2	Passive with extra features, rewritable.
Class 3	Communication is powered by the reader, microchip is powered by battery, rewritable.
Class 4	Active, can talk to other tags without the need of energy from a reader, rewritable.
Class 5	The tag itself is able to power and read Class 1-3 tags and behave like Class 4, rewritable.

Table 2. Advanced EPCglobal tag classes.

Specifications for these classes are in planning or design phase and were not publicly available at the time of writing this thesis.

3.2.1 EPC Class-1 Generation-2

The Gen2 protocol is the latest revision of the EPC Class 1 Specification. It specifies operation in the 860-960 MHz range, so the same specification can be used both in Europe and in the USA by implementing the matching protocol variation. Compared to Class 0 and 1, it offers higher performance and increased security. Its anti-collision methods prevent tags from interrupting each other when being in the range of the same reader. It has also support for environments where many readers are active simultaneously.

For security, Gen2 tags store an access and a kill password. If a tag has a non-zero access password then write access is only possible if the correct access password is supplied. Parts of the memory may be locked (i.e. made read only) by issuing a Lock command. The Kill command renders a tag useless, such a tag will not respond to the reader anymore. It is a permanent operation and can only be done by giving the correct kill password before. The communication from the reader to the tags uses basic encryption. There is no encryption in the other direction since the reader transmits with much higher power, compared to the power of the response coming

from the tags, so it is the signal transmitted by the reader that is more easily available to unauthorized “listeners”.

(Detailed description of Gen2 operation can be found in the *Class 1 Generation 2 UHF Air Interface Protocol Standard* and in *Glover, B. 2006*.)

3.2.2 Limits and regulations

The usage of radio frequency technologies fall under local regulations that may present serious limiting factors regarding speed and reliability. An example of this was the spectacular difference between the regulations in Europe and the USA. Before the EN 302 208 standard was put out by the European Telecommunications Standards Institute (ETSI), the frequency allocation available to UHF RFID uses was the mere 250 KHz band from 869.4 MHz to 869.65 MHz with the limit of 500mW ERP. Meanwhile, the US regulations allowed 4W EIRP from 902 MHz to 928 MHz This meant that in Europe a compliant UHF RFID system had to deal with decreased data throughput, shorter distances, and was less reliable. (*Eeden, 2004.*)

ERP and EIRP are used to estimate the coverage area of transmitters on the same frequency and to provide limits for the transmitted power, especially in built-in areas. *ERP* stands for Effective Radiated Power and equals the power submitted to the antenna multiplied by the antenna gain in a given direction or the direction of the maximum gain. In general the *gain* is the ratio of the signal output of a system to the signal input. The antenna gain is a property of the given antenna and can be found in its specifications and is given in dB or dBi. From the programmer's viewpoint the antenna gain is simply a value that must be set during the configuration of the reader to allow proper calculations of the transmit power level. This may be more important if high power is being used for increased tag reading distances and reliability because it must still be made sure that the local regulatory limits are not stepped over. With some readers there is no explicit way to set the gain – it must be calculated into the power level settings by the programmer. To convert from dBi to dB the formula `gain_dB = gain_dBi - 2.1` can be used. *dBi* stands for decibels relative to isotropic

and defines the antenna gain relative to an isotropic antenna. The isotropic antenna is a theoretical construct that evenly distributes power in all directions. *EIRP* stands for Effective Isotropic Radiated Power and it is the power that would have been emitted by an isotropic antenna to produce the peak power density observed in the direction of maximum antenna gain. 2W ERP equals to 3.3W EIRP (`p_eirp = p_erp * 1.64`) so the US regulations still allow transmission with a somewhat higher power than is Europe.

Sometimes with some readers the transmit power level settings must be given in units of dBm that is the power ratio of the measured power referenced to one milliwatt. For quick calculations the formula `val_dbm = 10*log(10, val_mw)` can be used. This means, to name a few important values, that 1 milliwatt is 0 dBm, 100 milliwatts are 20 dBm, 250 milliwatts are 24 dBm, 500 milliwatts are 27 dBm, 1 watt is 30 dBm, 2 watts are 32 dBm and 4 watts are 36 dBm.

The ETSI EN 302 208 standard allows the usage of the band 865-868 MHz with up to 2W ERP. This finally allows compliant UFH RFID systems to be more competitive with systems in other parts of the world in terms of reading speed, reading distance, and reliability. It makes the Listen Before Talk mode of operation compulsory. This means that before each transmission the reader should "listen" into the desired frequency band for some time. If any signal over a given level is detected then the band is already taken by another reader or other transmitting equipment. The reader should then choose another band or wait until the originally selected one becomes clear, and start over with the listening process. To be fully compliant with the latest standard the reader should be set to use the 865-868 MHz band and Listen Before Talk should be turned on. Correct transmit power level and antenna gain settings are also required to make sure that the power limit is not exceeded. In most countries of Europe the new regulations are already implemented, usually allowing the usage of the band 865.6-867.7 MHz with 2W ERP and Listen Before Talk. For example in Finland they are in place since February 2005.

Some UHF readers are able to operate in multiple regulatory environments while some support only one. In case of multiple environment support some may have easy configuration options to switch between the supported regulatory settings, while with

others the different options that are relevant to operation within the limits (e.g. which band to use, listen before talk) must be set one-by-one. Listen Before Talk is usually performed by using one of the normal antennas for listening for a short period of time. Besides this some readers have an additional antenna port and the antenna connected to this port will only be used for listening.

3.3 LF and HF RFID technologies

Low-frequency RFID uses the frequency bands 125-134 KHz and 140-148 KHz. It features short reading distances and slow transfer rates. It was and is widely and successfully used in animal tracking and access control. The maximum read range with passive tags (assuming suitable equipment) is about 50 centimeters. This is the technology that is mostly available to home and hobby users at the moment, for example the Phidgets RFID board is available from Phidgets, Inc. (http://www.phidgets.com) for CAD 65 at the time of writing and is able to read tags within 7 centimeters of the reader. However, similar HF readers are starting to appear at similar reasonable prices. It must be noted that the only functionality these simple devices usually provide is reading one tag at a time and retrieving the 32-64 bits long ID (and the communication often uses some proprietary protocol).

High-frequency RFID works at 13.56 MHz and it can be used throughout the world without any license on the frequency spectrum, unlike UHF. The data transfer speed is higher than LF and while the typical maximum read range is 15 centimeters, it can be typically extended up to 1.5 meters in a gate configuration where almost the entire gate serves as an antenna. This technology provides the highest storage capacity at the moment, for example our cheap credit card-sized sticker-type rewritable HF tags were able to store 80 bytes of user data and newer tags offer much more space. It does not suffer from electrical noise and does not have problems with metal, liquids, and other shielding material. Therefore it is the ideal choice in libraries (being able to read all the tags in a pile of books) and for security gate purposes. However when a large number of tags are passing through a gate at once it is almost sure that current HF readers will not be able to cope with all of them. Also, the more tags are present in the

field of the reader, the bigger is the possibility of phantom reads. This means that a non-existent tag will be reported by the reader, and such behavior could be reproduced with as few as five ISO 15693 tags by presenting them to the MP9210 reader at once. The reported phantom ID was sometimes real garbage while sometimes it only differed from other real tags' ID in some bits.

There are two major ISO standards in use today: ISO 14443 and 15693. While the latter allows a distance of about one meter, the former only provides about 10 centimeters but with a much higher data transfer speed. Besides these, there are proprietary technologies, e.g. MIFARE, that use the same frequency and are similar in some aspects to the ISO standards but they differ in reading distance, speed, and are not compatible. However, multi-protocol readers may support different types of tags.

3.4 Device interfaces

Most readers offer more than one communication method between the device and the computer. Most common are the serial and Ethernet interfaces. Some also have the possibility of using USB connection. Wireless connections, like WiFi and Bluetooth are not so wide-spread today but may become more popular in the close future as they simplify cabling and device placement problems. However, proper configuration may get more complicated in these cases as wireless technologies introduce another possible security hole in the system. It is enough to have a look at the seemingly never-ending discussions about wireless network security. Naturally, in case of an 802.11-based connection these issues can be quite easy to overcome, if the device is intelligent enough to support the latest wireless security technologies, but then again, it makes the device more complex, and it completely depends on the environment, whether strong security between the reader devices and the computer is really needed. If the place is isolated enough from outsiders, then unwanted spreading of the wireless signal may be less of a problem.

RS232 is the best known among the serial communication methods. It offers the most simple way of quickly connecting the device to the computer as such serial ports are

still readily available in most computers. The topology is strictly point-to-point and the speed and cable length are limited. Regardless of its age and limitations, the RS232 connection may still be appropriate for a system with low throughput, i.e. when the amount of data generated from tag observations is guaranteed to be small enough. The serial connection may also be sufficient for test-driving the reader in the beginning of the development phase when the main goal of engineers and developers is to get to know the device and its capabilities. However it is easy to see that scalability problems can easily arise so in general no new designs should rely on this kind of connection especially when using recent devices which may provide less and less support for RS232 even if the connector is there. For example while the reader protocol of our SAMSys devices had the same level of support for the serial connection as for the Ethernet one, this was not the case with the Caen A928/948, the protocol of which is influenced by EPCglobal's *Reader Protocol Standard* described later in this section. The asynchronous notification mode was simply unavailable with a serial connection (see *CAEN UHF RFID Readers Communication Protocol* p. 4), which is partly understandable since the handling of synchronous and asynchronous communication over the same channel can often be complicated and error-prone. This is discussed in more details in Section 5.5. With a sockets-based network connection the implementation of a separate channel for asynchronous notifications is quite intuitive, the reader can also listen for incoming connections on a port other than the one used for the control channel. While the communication on the control channel follows the usual request-response pattern, traffic on the notification channel is only initiated by the reader. With a RS232 connection there is exactly one channel and thus control and notification messages must be mixed onto the same channel. The designers of these readers seem to have opted for avoiding the more complicated programming model, and thus, users with a serial connection only must stick to the method of polling, i.e. the application must query the list of seen tags periodically from the reader.

Some readers, like our SAMSys MP9320, also have the capability of using RS485 connection. With this it is possible to create a multi-point topology that offers much higher transmission speeds for bigger distances with improved immunity to noise, when compared to RS232. It provided the ability of building a network from multiple readers even before Ethernet interfaces in readers became widespread. In most new

deployments this kind of connection is often ignored because of the availability of Ethernet connectors in almost every reader and because of the possibility of simply "throwing in" the reader into the existing network infrastructure.

3.5 Privacy considerations

As time is passing the upcoming RFID technologies offer better and better ways for securing and encrypting the communication between the tags and reader. The level of security always depends on the actual installation environment. This means that if the premises of a company or factory where the RFID equipment is installed are isolated enough from outsiders then there is most probably no need for heavy securing of the reader-tag communication. In our demonstration system, being a proof of concept project, no real security was implemented, for example someone could have easily rewritten the tags on the items, which would be totally unacceptable in a real production system. There the tags should be write-protected, the Gen2 access and kill passwords should be set, etc. For a reference on works related to security in privacy in RFID systems see the web page *Security and Privacy in RFID Systems* (http://lasecwww.epfl.ch/~gavoine/rfid).

There is a large on-going discussion about the privacy issues caused by RFID. Generally, the problem is that a tag in one's pocket or bag may get read without a person's knowledge and permission by someone who is standing nearby and has an RFID reader. With the basic approach (that was also used in our system) the tags attached to the products contain nothing but a 96-bit long value which is meaningless in itself. One needs access to the database to check what kind of product is the tag attached to. This is good from the privacy viewpoint because a third-party entity scanning for tags randomly will most probably not be able to do anything with the raw tag ID, even if the tag in someone else's bag got read. The disadvantage of this approach is that information sharing and co-operation with business partners is difficult. For example, in case of libraries and other rental services, the items (e.g. books) may get transferred to another unit for some time. If access to the database of the owner unit is not available then the tag on the item is useless, and a new tag must

be placed onto the item at its temporary location. Then again, this new tag will be considered useless for the owner unit when the item gets returned to its home location. Many plans and visions exist on how to enable RFID information sharing, usually involving central databases and unified data formats. For a data model that allows inter-library loans, see for example the document *RFID Data Model for Libraries: Finish Data Model. 2005*.

When the basic "own-tag-database" approach is taken, the ID uniqueness problem presents an increased risk. While it is guaranteed that two tags will never contain the same ID within the boundaries of the system, what happens when someone, accidentally or deliberately, walks before the reader with a foreign tag that holds an ID that is equal to an ID that is assigned to a tag in the system? If the tag gets read the system will think that the person holds the item to which the tag with the same ID is attached. The list of possible consequences is very long, depending on the system. For example, in a library when doing a check-in (the process of bringing back one or more books) an automatic system could be fooled to think that the book is really there while only a cloned tag is presented to the reader. On the other hand, in the accidental case the security gate in a shop may alarm and the innocent person, with accidentally a tag in his or her pocket unfortunately holding the same ID as the tag attached to some product on the shelves of the shop, may get accused of committing theft.

There are increasing fears about government, authorities, and criminals tracking people's consumption habits, movement, behavior, origins, habitation, political conviction, health status, etc. by scanning for tags that are embedded into or attached to products that are lying in one's pocket or bag. Meanwhile, the existing solutions to prevent unwanted reading of tags, like the blocker tag (which confuses readers by clever misuse of the tag protocol so that no tags will get read at all), shielding wallets, etc. are gaining more and more interest. However, with the current state of the (widely available) **passive** RFID technology it is quite hard to imagine that someone, armed up with an RFID reader, starts scanning people passing by from several meters. It is often problematic enough to get a good read ratio with tags attached to items that are intentionally passing through a UHF gate, not to mention looking for tags in the backpack of people who are moving unpredictably. LF and HF technologies have even more limited reading ranges. With evolving technology and spreading of active tags,

however, the possible read distances and reliability may get increased so the protection of personal privacy might become more important than ever.

4 SOFTWARE COMPONENTS

There is not much benefit in having a reader lying on the floor, some antennas mounted on the wall and walking around with tags while enjoying the beeps indicating that a tag was read. The reader should be connected to a host (a regular computer, a handheld device, etc.) on which some software is controlling it and processes the incoming observation data. In simple and small-scale systems this software can be one single application that communicates with the reader directly or by using some vendor-provided libraries. In a complex system there is usually at least one software entity between the device and the high-level applications: the middleware.

4.1 Reader protocols

A reader protocol specifies how an RFID reader and a host can communicate. This means specifying a syntax for messages (reader layer), how these messages are formatted to be suitable for transport (messaging layer), and how the messages are transported over some kind of a network, serial, etc. connection (transport layer). The number of possibilities is quite large, a reader could communicate with commands formatted as plain text, binary data, or XML, and this data could be transferred over a serial connection, in TCP packets, over HTTP (some recent readers even have a built-in web server), etc., not to mention the syntax of the commands which can be completely device-specific and proprietary.

4.1.1 The Reader Protocol Standard

EPCglobal's Reader Protocol Standard tries to present a uniform way for accessing and controlling an RFID reader. Since reader capabilities and features may differ, it does not require the implementation of all the described functionality. It provides an implementation for text and XML message formats and implementations of the transport layer for serial connection, TCP, and HTTP.

There are readers which do not fully implement the Reader Protocol Standard but some parts of their operation and protocol are inspired by some of the approaches and concepts described in the standard. For example the Caen A928/948 is such a reader, with a binary messaging layer (*CAEN UHF RFID Readers Communication Protocol*). Therefore it is wise to get to know some of the concepts from the standard. This is beneficial also when dealing with other EPCglobal standards and recommendations, for example the ALE specification (see Section 4.2).

Instead of just speaking about antennas, the standard introduces the concept of sources and readpoints. A readpoint is “any physical entity that is capable of acquiring data”. Besides antennas, this can mean any sensor, e.g. a bar code reader or an optical sensor. A source is a container for (among others) readpoints. Therefore more than one antenna can be grouped together to form a source. It is the source that generates the events that are sent as notifications, not the individual readpoints. For example, if a reader has three antennas attached, two of which are used to form a gate, and the third is placed somewhere else for other purposes, then the two readpoints corresponding to the “gate antennas” can be grouped together into a single source, and the third can be assigned to another source alone. Most applications will be interested in tags seen by the gate and will not care which antenna out of the two did actually read the tag. The grouping of the two or more readpoints into a single source matches this behavior very well. (see *Reader Protocol Standard* p. 34)

There are two independent logical channels defined for communication: the control channel and the notification channel. On the latter it is guaranteed that traffic will only be initiated by the reader while the control channel is used for request-response type

messaging. However, the implementation of separate channels may not be easy or may even be impossible in case of some transports. For example, when connecting through an RS232 serial connection there is no way to use more than one communication channel so both the control and notification channels must be multiplexed on the one available channel. This may cause headache on the host side since an asynchronous notification can take place at any time, probably after a request that has been issued but before the arrival of the matching response, the host must be able to handle all of these situations. In contrast implementation is simple and intuitive with TCP: the reader listens on two pre-defined ports for incoming connections and the host will connect to both simultaneously. While most readers today allow only one host to be connected at a time, the standard allows readers to handle multiple hosts at the same time. This, of course, makes things more complicated on the reader side but future readers are likely to support this. (see *Reader Protocol Standard* p. 19)

The simplest RFID readers contain almost no intelligence regarding the processing of tag observations. These readers do not maintain any state information, when a tag is read, a notification is sent immediately, then, since a tag is read many times even during a very short amount of time, another one, and so on, until the tag is removed from the field of the antenna. The host receives these raw tag observations and the host is responsible for converting these to higher-level events in which the applications are interested. For example, using the gate example again, if an application is only interested in getting a single notification when a tag is detected by the gate then flooding the notification channel with a high amount of raw tag observations is a waste of resources. If the reader is intelligent enough then data volume can be reduced by sending a lower amount of higher-level event notifications, for example one single event, which indicates that a tag appeared at the gate, and later another one, indicating that the tag is not “visible” anymore, would often be enough.

Such operation is called smoothing. It must not be confused with filtering which means that the reader should only send notifications about tags the ID of which match an application-defined pattern and therefore tags that are not interesting to the application will simply be ignored. Smoothing means that the amount of transferred data is reduced “by generating events when the the status of each tag changes”. (see *Reader Protocol Standard* p. 25-28 and *Glover, B. 2006.*) While earlier filtering and

smoothing was mainly the responsibility of the middleware as readers were not intelligent enough, most of these operations can now be done at the reader level and thus reducing the traffic also between the reader and the middleware (see Section 4.2 about RFID middleware).

4.1.2 CHUMP

The Comprehensive Heuristic Unified Messaging Protocol is the protocol used by SAMSys readers. It uses human readable plain text messages. While in the EPCglobal Reader Protocol Standard the message layers using a human readable format are recommended mainly for debug purposes and not for production use, these readers show that a relatively simple plain text protocol can also be sufficient. The protocol uses a single channel onto which synchronous and asynchronous messages are multiplexed, which is not surprising when taking into account that the protocol was originally designed for RS232 and RS485 connections. When using the Ethernet connector instead of serial nothing changes, the same text requests and responses are sent over a TCP connection, in fact these readers can be configured and tested without any additional software as it is possible to connect to the reader with a telnet program and issue commands manually from the keyboard.

In most cases the requests sent to the reader are in the following form:

```
}<command>;<checksum><CR><LF>
```

while the responses from the reader follow the same format with the difference of a `{` character used to distinguish between requests and responses. For example sending the request

```
}Rd;<checksum><CR><LF>
```

(see *[CHUMP2005]* p. 126 about the checksum generation algorithm) will result in one or more responses from the reader in the form

```
{Rd,d:<tag id>,t:<tag protocol>,r:<repeat count>,
    n:<antenna port number><CR><LF>
```

In fact, this one command in itself may be sufficient for implementing the poll pattern; one should just issue an `Rd` command periodically and process the response.

There are a few other commands, e.g. for tag writing, locking, killing, user data reading, etc. The values of configuration registers can be changed with the `Cr` and `Cw` commands. All the reader settings can be configured via changing the corresponding bits on the given register. There are about 20 registers that may interest the user and that are for reader configuration, plus there are some tag protocol specific tag configuration registers. *[CHUMP2005]* contains the description of the meaning of the specific bits written to the registers. For example, setting bit 0, 1, and 2 all to one in the GCW (General Configuration Word) register will switch the reader to continuous mode, that is, when a tag is read the raw observation is sent to the client immediately. It is possible to order the reader (by setting bit 16 in the PCW register) to collect the observations in a queue and only report the contents on a specified interval boundary. Tag filtering can be enabled by setting the corresponding tag configuration registers, so it is possible to only have those tags reported, the ID of which (or a part of their ID) matches a given value. Besides this simple queuing and filtering mechanism there is no other support for converting the raw observations to higher-level tag events, it is left to the application or the middleware. Nevertheless, the simplicity and power of this protocol is still quite impressive.

4.2 Middleware

The main three tasks of RFID middleware are the following (*Glover, B. 2006.*):

encapsulating the applications from device interfaces,
converting the raw observations to meaningful, high-level events,
providing an interface for managing RFID readers.

Point two may look familiar from Section 4.1. This comes as no surprise since it refers to the same filtering and smoothing processes that were described there. New readers get more and more intelligent and many tasks that were traditionally the responsibility of software applications or the middleware, between the reader and the applications, can now be done on the device level. However, each reader has different features and capabilities, often with significantly different protocols, so RFID middlewares are here to stay for some time, not to mention that they offer much more than just the first two points from the list above. Middleware can support reader discovery, monitoring,

management, may provide service-oriented services (using standards, e.g. Web Services), and remote provisioning, just to name a few of the common features (*Glover, B. 2006.*).

ALE, which stands for Application Level Events, is an application-level interface "through which clients may obtain filtered, consolidated EPC data from a variety of sources". It is most of the times implemented in the middleware but it could also be deployed in a reader (and thus the applications on the host would use the ALE-compliant interface provided by the reader instead of the presumably lower-level reader protocol). It provides a standardized format for reporting tag data and this will be independent of the lower layer details, i.e. how the information was gathered and processed. The sources (readers, antennas, other sensors) are abstracted into the concept of logical readers. A logical reader can be thought of as a container for multiple source objects (here source refers to the concept of sources in the Reader Protocol Standard, see Section 4.1). This way, changes at the physical level (e.g. changing the number of antennas that from a single gate) will not affect the applications. For detailed description see *Glover, B. 2006.* and the *Application Level Events Standard.*

5 IMPLEMENTATION OF THE DEMONSTRATION SYSTEM

The goal of the demonstration system was to provide a basic implementation of a simple, automatic, intuitive, staff-less check-out environment using UHF RFID technology. The goods equipped with UHF Gen2 tags are collected in the customers shopping cart and the ID stored on the tags is read when passing through the check-out gate. The tags store no user readable information other than the ID to prevent the raising of serious privacy concerns. After passing the gate, the customer sees the information about the contents of the cart on a screen and identifies himself/herself with his/her customer card, if satisfied. The customer cards are HF RFID tags that store nothing but the ID. The last step in the demonstration is to print a receipt of the purchased goods.

5.1 Architecture

Physically the system consists of an UHF RFID reader with one to four antennas that form the gate, a HF RFID reader with one antenna (either integrated or external) that is used for customer identification, UHF tags that are attached to the individual products and are programmed with unique IDs, HF tags that represent the customer cards and are programmed with unique IDs, a computer connected to the readers, a screen which serves as the information display for customers, and a printer. Uniqueness here means that the tags must have such IDs programmed that are different from each other at least across the system. It is not guaranteed that there cannot be another tag somewhere in the world with an ID that is also being used by a tag in our system.

The main requirement for the system was to be able to do continuous operation without any interaction. There is no input to the system at all except for the notifications about tag reads coming from the RFID readers. However, the customer must always know what is going on, no doubt should be left about what state the system is in.

On the software side the system consists of a database server, a server application that provides transparent and unified connection to multiple RFID readers for multiple applications (this component is something like a RFID middleware but does not qualify as a real middle as it is explained later), and the end-user application that is used for the actual demonstration and provides the customer with visual and audible information while controlling the readers through the “middleware”, querying information from the database, and printing receipts.

At the beginning of the design phase the use of a real middleware product (e.g. the BEA WebLogic RFID Edge Server) was seriously considered but the idea was dropped due to licensing costs and to the fact that such a relatively simple system does not need such a relatively large component. Also, spending more time on low-level

issues (learning how to use the given devices from own applications without relying on any other product) helps obtaining a more clear view of RFID technology and how it works.

5.2 Use cases

The system is quite simple from the use case viewpoint as the users (the customers) have only one possible task: walk through the gate and complete the checkout process. Definitions of some expressions used in the use case and throughout the description of the system are as follows:

Item tag: An EPC Class1 Gen2 UHF tag.
Customer card tag: An ISO 15693 HF tag.
Gate reader: An UHF RFID reader capable of reading EPC Class1 Gen2 tags with multiple external antennas.
Customer card reader: A HF RFID reader capable of reading ISO 15693 tags with one integrated or external antenna.
Tag ID: A 96 bits (in case of UHF) or 64 bits (in case of HF) long binary value that is processed and stored in the database as a string containing this value in hexadecimal format.
Purchasable item: Any item or product that has an item tag attached to it.
Customer card: Some kind of a plastic card containing a customer card tag. For demonstration purposes HF tags with the appearance of a usual plastic card were used.
End-user application: The application that the customer sees on the screen during the check-out process. Informs the customer about the quantity and price of items found in the cart. When no more new purchasable items are discovered, it shows the total price and prints a receipt. May also play sound effects for better catching of attention when there is a misuse in the system.
Checkout: The process of putting at least one purchasable item into the field of the gate, identifying the customer with a customer card, and finally getting a printed receipt about the purchase.

Field of the gate: The area covered by the antennas attached to the gate reader. Item tags outside of this area should not be read.

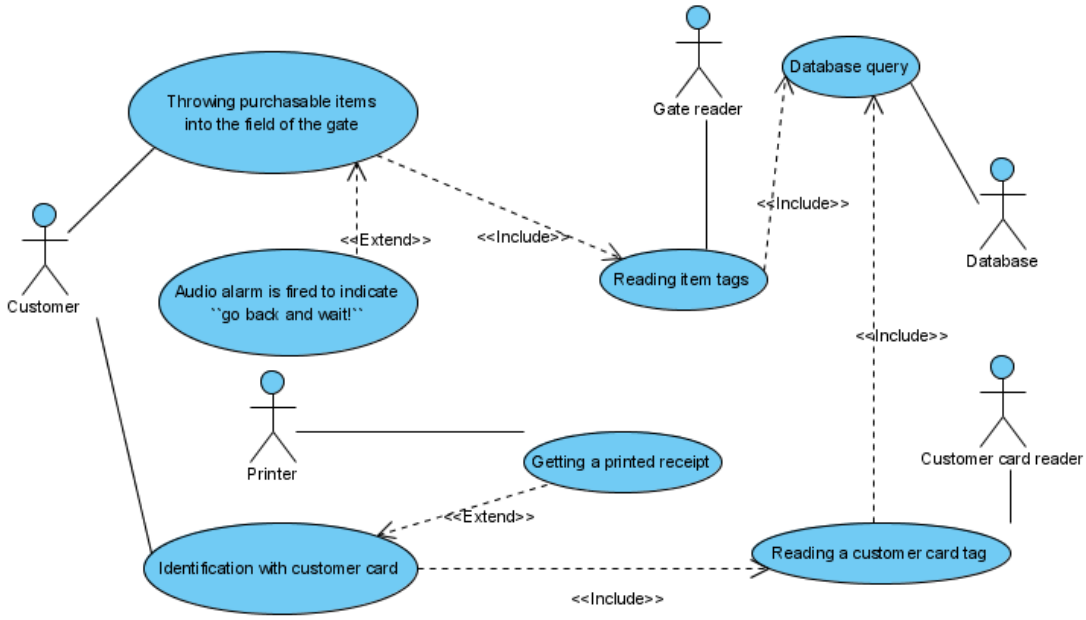

Figure 5. Use case diagram for check-out.

The use case is as follows:

Use case: Check-out.

Frequency: Very high.

Usability requirement: The user must know all the time what is going on.

Preconditions: The system is in the *waiting for customer* state, no previous check-out is in progress.

Description: The customer puts one or more purchasable items into the field of the gate. [Exception: A previous check-out is in progress] When the gate has discovered all the item tags, the customer is asked to identify himself/herself with a customer card, this is done by having the customer card tag read by the customer card reader. [Exception: No customer card is shown]. Finally a printed receipt is received from the printer. [Exception: Printer problem]

Exceptions:

1. A previous check-out is in progress: The system is still handling the previous customer, no item tags must be present in the field of the gate at this point to prevent confusion. The exception occurs when reading of the previous set of item tags

(purchasable items of the previous customer) is finished and the system is waiting for a customer card. The result is an alarm and the item tags that have just been read are ignored.
2. No customer card is shown: No customer card tag is read by the customer card reader for a fixed amount of time. The purchase will be canceled, the system will be ready to process the purchasable items of the next customer.
3. Printer problem: For example out of paper. The customer gets no receipt.

5.3 Hardware components

The UHF reader was a SAMSys MP9320 v2.8e-2, the product of SAMSys Technologies Inc., a company from Canada that was acquired by Sirit Inc. in April 2006. This device has four external antenna connectors as well as a dedicated fifth port for an antenna that is used only to perform improved listen before talk operation. This latest revision of the device supports operation in ETSI EN302 208 regulatory environments, this means using the 865-868 MHz frequency range, performing listen before talk, and using a maximum power of 2W ERP. It is capable of reading EPC Class1 Gen2 tags, after a brief experiment period with some ISO 18000-6B tags it seemed obvious to go for Gen2 because of its better performance. It provides RS232, RS485, and Ethernet connections. For application development a Java and a .NET library is provided.

It is easy to see that the HF reader is not a compulsory part of the system as its work can be done by one antenna of the UHF reader, especially when one takes into account that at the actual public presentation of the system the gate only consisted of one antenna and the three other ports were not used. However, by not relying on the gate reader, performance of reading item tags is increased since no time is spent on multiplexing multiple antennas. Another advantage is the availability of HF tags embedded into plastic cards, to the end-users these are not even distinguishable from a regular card. The limited reading range of the proximity HF reader actually presented another advantage: no worries about customer cards being read from far distances, i.e.

no unexpected reads of a tag sitting in someone's pocket who is standing near the reader by coincidence.

Figure 6. The SAMSys MP9210 HF reader.

The MP9210 reader has one integrated antenna, the maximum reading distance is about 15 cm. In the demonstration system it was used with ISO 15693 tags, besides that it supports tags using Philips' (“I-Code”) and Texas Instruments' (“Tag-It”) protocols. It provides RS232 and a RS485 connections. With a separate concentrator module, Ethernet may also be available, according to the manuals. Although the feature was not used in the demonstration, both readers are able to write and reprogram tags during normal operation. The reader protocols used by the MP9210 and MP9320 devices are virtually the same. This was a big advantage during the implementation of the system because it was possible to reuse most of the code for managing the low-level layer of connectivity. The only real difference was in the physical connection as the MP9210 does not have an Ethernet interface.

The antennas were circularly polarized UHF antennas. Some tests were done with antennas having linear polarization in the beginning but they got retired later because of the better area coverage of circular ones which was decided to be more important in this application. However due to the lack of the bigger read distance provided by the linear antennas the transmit power level had to be set higher which in consequence caused more reflections and a wider, often unexpected spread of the signal, resulting in unexpected reads of tags that were definitely not in the area of the gate.

5.4 Building the system

Having all the software perfectly written and tested does not guarantee anything – if the RFID reader fails to read the tags then the whole system is useless. Naturally the possibility of missing a tag is always there, at least with today's technology, but the probability of this should be reduced to the minimum. The problem is that there is no golden rule to achieve this, current RFID installations often involve the trial and error method, i.e. the engineers set up a test environment and start experimenting with antenna placement, the angle of antennas, distances, power levels, and so on, in order to get the best possible reading rates with the smallest number of misses. However, all these efforts may become a waste of time because the laboratory environment and the final place of installation can be very different. UHF signals tend to have problems with metal surfaces, liquids, even the human body, and when transmitting with higher power levels reflections occur which can result in unexpected spread of signal into areas which are far away from the normal coverage area of the antennas. The document *[TI2005-3]* covers the UHF RFID installation problems that take place most often and some of the most widely accepted best practices that try to solve these. As it will be discussed in Section 5.4.2 most of these problems were met during development.

5.4.1 Settings up the readers

The cabling work is usually simple, in some environments the length of the cables may cause some problems if there are bigger distances between the antennas and the reader or between the reader and the computer. With antenna cables it is important to remember that the longer the cable the bigger the cable loss (the degradation of the signal while traveling through the cable) is. With our system there were no such issues, both in the laboratory and at the exhibition site the physical setup went without problems, with minor issues like how to hide all the cables from visitors as much as possible.

A minor complication here was caused by the network configuration of the UHF reader. While the MP9320 was able to acquire a network address automatically via DHCP and this indeed was the preferred way in the laboratory network, this is generally a bad idea because the address assigned to the reader is not known (the device cannot tell it to anyone if nobody is able to connect to it) and even if it gets traced somehow there might be no guarantee that the same address will be assigned the next time (then again, this could be solved by assigning DNS name to the reader and keeping it up-to-date with the address, etc.). The problem may seem trivial, but sometimes it is easy to forget that the reader should be treated in the network like any other server computers.

The solutions for setting a statically assigned network address were different in the two tested UHF readers: the Caen device has a command in its reader protocol that is used to change the address while the MP9320 relies on an external utility named Digi Device Discovery. The obvious advantage of the first solution is the ability to set the network address while connecting via RS232, for example. The second method draws on existing tools that follow the pattern of the often ugly way of changing network settings of Ethernet connection enabled devices (e.g. network printers) that do not have a proper user interface to do such changes on the device itself without external help from a computer.

In case of our reader the procedure was the following:

1. Connect to the reader with a cross-over (patch) cable.
2. Run the discovery tool which will do an Ethernet broadcast to which devices that have a Digi Connect module (an embedded network interface by Digi International, Inc.) will reply.
3. Select the MP9320 device from the list of the IP and MAC addresses belonging to the devices that have been seen, configure its network settings, and restart the device.

Step one might seem unnecessary, but consider the following case: in the laboratory network there were lots of other devices with the same kind of network module and therefore it gave some headache to find out which entry in the long list belongs the

RFID reader. Identifying it by the actual IP address was not possible since according to factory defaults the reader acquired an address via DHCP and this address was both unknown and very similar to addresses of other devices. The solution was to connect the reader and the computer directly by using a cross-over cable so that no other devices could be discovered and cause confusion.

5.4.2 The gate

One of the first questions, regarding retail, that may arise is that what kind of RFID, UHF or HF should be used? If UHF is the choice in the manufacturing environment (this is almost always the case) then going for HF at the retail level would be a bad idea, especially if a country- or world-wide co-operation is needed regarding tagging between suppliers, the shopping units of the company, and perhaps other retailers. Also, UHF is better suited for handling a bigger amount of tags at once, from a greater distance, so it may seem more ideal for gate-type installations, at least today. Yet it has some unsolved difficulties by nature: the deterioration of the signal with metal surfaces, liquid material, and the human body. Active tags may bring more reliability, better read ranges, and more intelligence, the downside is the price and the need for a power source, both of these make them unsuitable for putting them on retail products in mass amounts at the moment.

The most critical part of the system is the gate itself. If a tag attached to a product lying in the shopping cart does not get read then the whole system may be considered useless. Unfortunately, achieving perfect read ratio is almost impossible with the current state of technology. More precisely, a ratio very close to 100% may easily be possible in the traditional supply chain environment model, i.e. when boxes with RFID tags attached to one or more of their sides (and thus the tags will face the antennas in front of which the boxes pass by) are traveling on a conveyor belt. In contrast, when the tags are attached to arbitrary products that are carelessly thrown onto each other in a shopping cart or a bag then the missed tag ratio is usually much higher. In the following few points a summary of the things learnt during the experimentation with the gate is given. Most of the conclusions match the widely

accepted UHF RFID installation recommendations. Note that many of the problems and limitations come from the nature of UHF signals, and thus, they may not be true for other RFID technologies, like HF.

The shopping cart

The shopping cart used was a regular one that can be found in every supermarket and it was made of metal material. If a product was placed accidentally in a way that the tag was touching (or was very close to) the sides or the bottom then it was almost always sure that the tag would be missed by the reader. To reduce the occurancy of this case a cardboard backing was installed at about one centimeters above the level of the bottom of the cart. This way the tags were never touching the bottom metal surface even when they happened to be facing down.

The angle and distance of tags

The antennas were circularly polarized so the orientation of the tags was not so critical as with linear polarization. Still, throwing products randomly into the cart (like it is done during shopping) can easily produce cases when the tag is in a very unfavourable position regarding the antenna. Since forcing the customers to place the products in such a way that the tags will be positioned correctly is impossible, there is no clear solution to the problem at the moment, at least not with the relatively cheap sticker tags. A similar problem occurs when two tags get too close to each other; one or both might not get read. Again, the probability of this is not really high in case of using a shopping cart but it can happen, and when using a bag or a basket instead of the cart, the probability of angle and distance problems gets increased radically.

Shielding

Generally speaking metal and liquid material absorbs the UHF signals or causes strange reflections. Additionally the human body shields very well, during our tests it seemed to be impossible to read a tag which was touching parts of the human body. Therefore the use of a generic UHF RFID gate for security purposes has some serious concerns with the current state of technology. In contrast, HF RFID does not suffer from these problems and if the smaller reading range is acceptable then it may be a better candidate for such purposes.

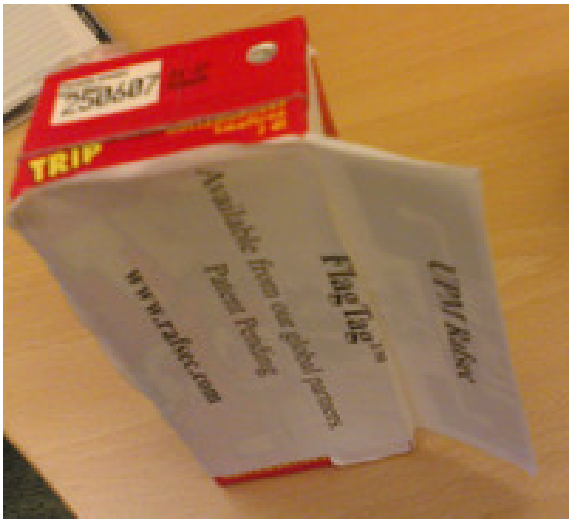

Figure 7. A working albeit arguably not very beatiful solution for tagging products containing liquids.

We had some experiments with adding some pieces of metal foil around the antenna to prevent unwanted spreading of the signal before and after the gate. It is a fatal error if tags sitting in the shopping cart of the previous or next customer get read. The tests did not bring any good results and this simple solution did not prove to be reliable, especially since with high power levels reflections will occur anyway and this happens far away from the antenna so limitation of the coverage is definitely not guaranteed. Trying weird solutions, like covering almost the entire gate with shielding material, did not seem to be a good idea, primarily for aesthetic reasons. It can be said as a general rule that enough attention must be spent on determining and marking (also physically on the floor) the coverage area and it might be a good idea to extend this with a "safety" area around the gate. It can be then assumed that there will be only one set of tags (i.e. items belonging to the same customer) inside this area but it raises the question how this could be guaranteed and enforced in a real supermarket environment.

Antenna placement

Regular UHF RFID gate installations usually have one or two antennas mounted on both sides, facing each other. It is often a good idea to mount them in a slight angle which results in a better performance when the tags arrive in various positions, not only parallel to the antenna.

Figure 8. The layout of a regular UHF gate, with two antennas on both sides.

In our case this layout was not sufficient as the metal material of the shopping cart caused unacceptable reading ratios when the antennas were mounted horizontally. The obvious way to go was to mount at least one antenna vertically, facing straight down, and optionally one or two others that are facing not straight down but are rotated by about 45 degrees. In the end it was decided to use only one antenna (facing straight down) at the final presentation.

Figure 9. Snapshot of an early testing stage for the gate.

5.4.3 Testing

The software suite coming with our UHF reader was not too rich and was inconvenient for making rapid changes to reader settings (e.g. the transmit power level). This was very unfortunate since the configuration settings had to be tweaked very often while experimenting with antenna and tag positioning. To make the testing process easier a test application was developed. It connects to the server application (described in Section 5.6) and provides access to all the reader settings supported by the server and the driver for the particular reader on a simple and intuitive graphical user interface. It also supports basic tag writing. Since our rewritable Gen2 tags came from the factory with the same ID, each one of them had to be reprogrammed manually. The test program made this process easier: after specifying a 96-bit value and writing it to the first tag, the value was increased by one automatically, the operator put the second tag into the field of the reader, if the write operation succeeds then the value is increased again, and so on. The raw tag observations are shown in a grid where the count column can be used to decide if the given tag and antenna position is acceptable, if the value is very small then the read should be considered an accidental read which means that there is absolutely no guarantee that the tag will get read when going through the gate with the shopping cart again.

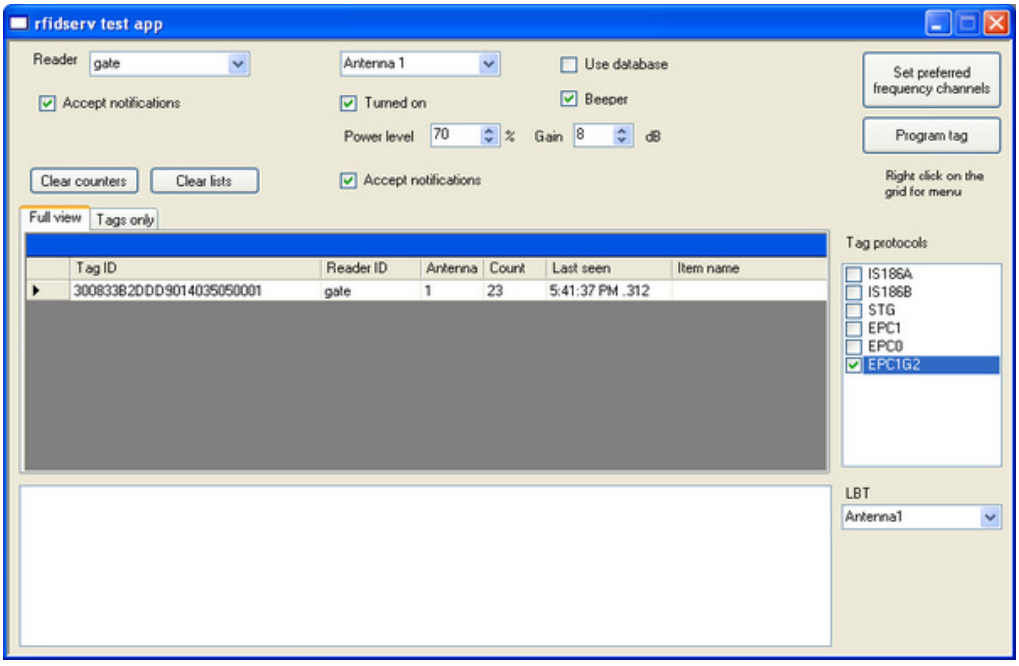

Figure 10. The test application, showing the settings of a SAMSys MP9320 UHF reader.

5.5 Common programming issues in communication

When a program needs to communicate with an RFID reader directly, i.e. without relying on some service (e.g. some kind of a middleware) that stands in the middle and hides the details of how the communication to the reader is established and managed, the programmer has two choices. One is to use an API (Application Programming Interface) provided by the vendor of the reader most often in form of a – either statically or dynamically linked – library. This is the only choice if the reader protocol is unknown (proprietary to the vendor or proper documentation is not available). Using a vendor-provided API makes the life of the developer much easier as the application does not need to care about how the connection (network, serial, USB, etc.) to the reader is maintained, and how the data flow from and to the reader is formatted and managed. The other way is to make a direct connection to the reader, for example by using sockets in case of a network connection or accessing the serial port directly in case of RS232. In the demonstration system the latter solution was chosen since the provided APIs did not prove to be mature and well-documented. Before discussing how the implementation was done two important areas of application development are be covered: multithreading and networking using sockets. They are both unavoidable and some kind of concurrent processing is very often required even with high-level APIs because the events (e.g. tag notifications) and data will still be passed to the application in an asynchronous manner. A well-designed API can hide the low-level details but the processing of tag-related events in the applications is still done using the poll or the publish-subscribe pattern.

5.5.1 Multithreading

By nature the RFID tag observations occur asynchronously without any respect to the state of any process running in the system to where the notifications about tag events are submitted. In the case of a computer program, that wants to receive and process tag events, this means that a notification may occur at any point of the program execution. If a reader would strictly forward the raw observations to the host then

handling the incoming data with one single traditional non-concurrent (single-threaded) application would be impossible, in this case a separate process or service (at least a simple middleware-like application) must be used to collect and store the incoming observation data until at some point it is passed to the high-level application. Fortunately some kind of queuing is implemented in almost all readers so applications can use a synchronous poll operation to query the list of observed tags when they want. In addition, some readers also offer filtering and smoothing already at the device level, see Section 4 for a description of these operations. The problem with this method of doing periodical polls is that it can be inefficient regarding the performance as the user experience may get degraded if the poll operation is time consuming.

The main data processing part of such an application will often have a structure like the following:

```
while true do
    check for exit condition
    do something
    poll the reader to get the list of observed tags
    do something, check if interesting tags were observed, update the
        user interface accordingly, etc.
```

In many cases it would be advantageous to avoid this model and keep using a seemingly traditional program structure while the incoming asynchronous notifications are processed concurrently. This is relatively easy to do on modern computer systems and many widely used programming languages have support for multithreading built into the languages or at least external libraries are available for implementing multithreaded operation. This way the program can be split into multiple tasks that are executing simultaneously from the programmer's viewpoint. Of course, real simultaneous execution is only available on systems with multiple processors, however, if one single processor is switching between the tasks fast enough then the illusion of simultaneous operation can be kept. The processing of any kind of incoming data (e.g. notifications coming from an RFID reader) can be done in a separate thread while the main thread is free to do its usual activity. This is especially useful if the input from an external device is not always available and blocking (waiting until data is available) is involved, for a good example on this see

Section 5.5.2 where socket-based network programming is discussed. The following parts of this section describe the aspects of multithreading that are most important for implementing a software interface to an RFID reader. The actual threading model and API can vary from system to system, most of the examples will be given for C# and the .NET Framework here since this was the environment used for all the software in the demonstration system.

In a multithreaded environment each process (a running program) has at least one thread - if it is the only one then the process is single-threaded- but it can have more. The threads of the same process can easily share data between themselves, by using global variables, member variables, etc., unlike different processes which have no access to each other's data and thus more complicated inter-process communication methods must be used. In some operating systems (like Windows NT-based products) the difference between the cost of creating processes and switching between them is considerably higher than using threads. In Unix-like systems the difference is smaller.

In a procedural language environment a new thread is usually created by calling a function provided by the operating system (if threads are supported on that level) or by some library (in case of user-level thread support) and passing a function pointer to it, this specifies what routine should be executed in the new thread. The function that created the thread will continue to run and “at the same time” the other function is also executed after the new thread has been created and started. More precisely, after the second thread is created and is scheduled for execution, the scheduler will execute some instructions belonging to one thread, then switches to another one, and so on. How and when the context switch occurs depends on the scheduling algorithm and other factors. Early versions of Windows used cooperative multithreading where a thread must explicitly yield (to indicate that the processor can be “given” to another one) as opposed to preemptive multithreading where the context switch can occur at any time.

Usually in a thread some activity will be repeated until an end condition (for example the application is exiting) is met. Proper termination of threads is important as the application will not exit completely until it has running threads. The simplest way is to use a flag variable to indicate if termination is in progress, the thread checks this value

in each iteration and breaks the loop if needed. However, when the flag is set by for example, the main thread the worker threads can be at any point of execution and thus will continue to do their work until execution reaches the point where the flag is checked. Blocking I/O operations also cause problems since the exit flag will only be checked after the operation has completed. The obvious solution to this is to use non-blocking calls which unfortunately tends to make development more time-consuming and error-prone. Another issue is that it is desirable to yield somewhere to avoid high CPU loads. This usually means calling a blocking I/O operation or the Sleep method (when this method is called the thread will not be scheduled for execution for the amount of specified time).

The basic structure of a program following this simple pattern is very often similar to the following:

```
Thread function:
1    while exitflag=false do
2        if there is data to read then read and process it

Main function:
1    initialize resources
2    create and start the worker thread
3    while exitflag=false do
4        do something
5        if exit condition is met then exitflag:=true
6    clean resources up
```

It is easy to see that this design has the possibility of a fatal error due to problem one described above. When the main function decides to exit the application, it sets the exit flag, breaks its main loop, and does the final cleanup operations (e.g. closes files and sockets, deallocates memory, etc.). We will now assume that the worker thread is using these resources during its operation in the *while* loop. Unfortunately, very negative things may happen if the worker thread does not have a chance to get to the point of the exit flag check before the main thread starts the cleanup. Accessing fully or partially closed and dismissed resources is never a good idea. There are at least two possible solutions, the first involves a simple check before the cleanup:

```
6    while worker thread is alive
7        Sleep(1)
8    clean resources up
```

The second uses some kind of synchronization facilities provided by the language or the system libraries.

```
Thread function:
1    while exitflag=false do
2        monitor.Enter
3        if exitflag=false then
4            if there is data to read then read and process it
5        monitor.Exit

Main function:
1    initialize resources
2    create and start the worker thread
3    while exitflag=false do
4        do something
5        if exit condition is met then
6            monitor.Enter
7            exitflag:=true
8            monitor.Exit
9    clean resources up
```

Here a theoretical monitor object is used that guarantees that only one thread can execute its instructions between the Enter and Exit calls “simultaneously”. If a thread calls Enter then all other threads that also call Enter on the same monitor will be blocked until the thread that owns the monitor calls Exit. Similar behavior can be achieved with synchronization primitives like semaphores, mutexes, and critical sections. In the modified example it is now guaranteed that step 4 is not executed after *exitflag* is set to true. The additional check in step 3 of the worker thread is needed because of the possibility of flag change between steps 1 and 2.

It is worth noting that in most environments there are explicit facilities to terminate (or at least request the termination of) a thread from another one. In C# and Java an exception will be raised in the target thread and sooner or later this will most probably

lead to the termination of it. However this kind of “aggressive” interruption is considered bad design by many, and it indeed has some danger, depending on the language and libraries, for example it can lead to resource leaks if the part of code that is being executed is not prepared for such “sudden” termination. See the description of *Thread.Abort* in the MSDN library for possible caveats under the .NET Framework.

Proper synchronization is very important when used shared data structures in multiple threads. Read operations are usually safe in this regard however in object-oriented languages more attention must be paid when calling methods of classes and class instances because sometimes it is difficult or even impossible to determine whether the given method does any modifications to instance or class data or is it safe to call from the more than one “simultaneously” running threads without any synchronization. For example, it might be easy to overlook that iteration on collections provided by the .NET framework is not a thread-safe operation as another thread may modify (add or remove elements) the collection instance meanwhile. In the following example the thread that manages the incoming data from an RFID reader will notify multiple observers when it receives a notification from the reader:

```
Receiver thread:
1    while exitflag=false do
2        monitor.Enter
3        if exitflag=false then
4            if there is data to read then
5                read(tag_info)
6                monitor2.Enter
7                foreach o in observer_collection do
8                    o.notify(tag_info)
9                monitor2.Exit
10       monitor.Exit
```

Naturally any other methods that modify *observer_collection* must call Enter and Exit on *monitor2* before and after the modification. The example is in fact not RFID-specific and provides an example for implementing the observer design pattern to provide asynchronous notifications to various subscribers about incoming data. A real implementation may, of course, face some unexpected difficulties like performance problems (if the *notify* methods do not finish fast enough) and cross-threading issues,

e.g. if the activity done within the *notify* methods involves updating a user interface made with the Windows Forms library of the .NET Framework, then the operation may fail or cause unexpected behavior because methods and properties of the user interface components are expected to be used from the main thread; while in this case the *notify* method and anything else it calls will run on the receiver thread. A solution, without using any additional support from the .NET Framework, would be to put the notification data into a queue (which is properly synchronized, of course) in the *notify* method and periodically extract the data from the queue in the main thread and update the user interface from there.

Unfortunately, after starting to use the synchronization facilities the programmer must usually face new difficulties, the most important being the need for avoiding deadlocks. A deadlock occurs when two or more threads are blocking and the condition under which they could continue is never met because they are effectively waiting for each other. A pattern for creating a deadlock can be as simple as the following snippet:

```
Thread 1:
    monitor1.Enter
    ...
    monitor2.Enter
    ...
    monitor2.Exit
    ...
    monitor1.Exit

Thread 2:
    monitor2.Enter
    ...
    monitor1.Enter
    ...
    monitor1.Exit
    ...
    monitor2.Exit
```

When thread 1 reaches the *monitor2.Enter* statement it will block infinitely because thread 2 will never call its *monitor2.Exit* since it will block on *monitor1.Enter*. While

it might seem easy to avoid incidents like this, it can be quite complicated to realize potential deadlocks in a bigger piece of code and problems often remain unnoticed until suddenly the program starts to behave “strangely”. Also proper attention must be paid to the handling of synchronization facilities, for example a missing *monitor.Exit* call in the above examples would be a fatal mistake and would cause a deadlock. It is easy to forget to properly release every lock, mutex, semaphore, monitor, etc. when a function has many exit points. Fortunately recent and popular programming languages like Java and C# usually have some kind of a synchronization construct built into the language itself.

Other possible questions regarding the example for processing incoming data are the following: how can the data availability be detected (without blocking) and where should or can we yield? In the demonstration system the following approach was taken: if there is no input available then the thread blocks until data arrives or a short amount of time has been elapsed. In either case the execution continues and if there is data then it is read and processed. When the additional activities (like checking for exit condition) are done the process is repeated. In case of network communication with sockets this is very easy to implement if the system has a *select*-like facility, see Section 5.5.2 for discussion on this.

In C# thread creation is done by creating a new instance of the *Thread* class and passing a *ThreadStart* delegate to the constructor, this specifies the method to be called when the new thread started and is scheduled for execution. For easy synchronization the language includes the *lock* keyword. When a block is marked with this keyword the block will be treated as a critical section and a mutual-exclusion lock will be acquired for the specified object by calling *Monitor.Enter* and *Monitor.Exit* where *Monitor* is a class that cannot be instantiated and provides some class methods. The behavior is the same as with the theoretical monitor object used in the pseudo-code examples above. When execution in a thread gets to the beginning of the marked block, *Enter* will be called to acquire a lock on the given object (it does not matter what type it really is, for these kind of purposes only the reference itself is used). If this is successful then the statements of the block are executed and it is guaranteed that *Exit* will be called whenever the execution leaves the block, even when an exception occurs, so the programmer need not care about releasing the lock at all possible

leaving points. If some other thread already holds a lock on an object then other threads calling *Enter* with the same object will block until the lock is released and then one of them will be allowed to continue. This construct makes it relatively easy to synchronize access to shared resources.

The .NET Framework offers lots of classes and features related to threading but for the purposes of the demonstration system the basic thread handling operations (like creation), the *lock* keyword, the *ManualResetEvent* class, and the asynchronous versions of some socket methods provided enough multithreading support. The *ManualResetEvent* class was needed when a thread had to wait for some specific condition to come true (e.g. waiting until some operation in another thread finishes) before it could continue. To achieve this, the thread sets the *ManualResetEvent* object to unsignaled state, and calls the *WaitOne* method which will cause it to block until the object becomes signaled. Another thread can just simply call the *Set* method to set the object's state to signaled. The following piece of pseudo-code is taken from the module that handles all the communication to and from the MP9320 reader. (Details about the protocol used by this reader can be found in Section 4.1.2 and in *[CHUMP2005]*) It is a simplified outline of how the receiver thread, that handles incoming data from the reader, works.

```
Receiver thread:
1    while true do
2        try to fill buffer until a newline character is read
3        if a full line is available in the buffer then
4            if it is a tag notification then
5                process data and notify observers
6            if it is a response then
7                resp_event.Set

SendRequestAndWaitForResponse:
1    resp_event.Reset
2    SendCommand(...)
3    resp_event.WaitOne
```

Since both the asynchronous notifications and the commands following the request-response pattern are multiplexed on the same channel the receiver thread cannot wait

until a response is received for the previous request, since notifications may arrive at any time and they must be processed as soon as possible. Instead, the function that issued the request and is executing in another thread will block by using a *ManualResetEvent* object and will only continue when the receiver thread sets the object into signaled state at a later time (when the proper response is received). The *SendRequestAndWaitForResponse* method is obviously not thread-safe but this is not a problem because in the higher layers of the server application proper synchronization is applied to all methods that access the reader (i.e. they send requests to it).

Besides the "traditional" collection of threading related methods, the .NET Framework offers other ways to achieve asynchronous and concurrent operation, one such way is the use of asynchronous delegate invocation. Instead of invoking the delegate directly, the *BeginInvoke* method is used which will return instantly. The method assigned to the delegate will be called from a separate thread and the calling thread is free to do anything else meanwhile, of course it should be notified somehow when the asynchronously executed method finished. A simple example can be something like the following:

```
delegate int D(int i);
void Callback(IAsyncResult ar) {
    D del = (D) ar.AsyncState;
    int result = del.EndInvoke(ar);
    Console.WriteLine("finished, result is " + result);
}
void DoAsyncCall() {
    D del = ...
    IAsyncResult ar = del.BeginInvoke(1234, Callback, del);
    while (!ar.IsCompleted) { ... Thread.Sleep(1); }
}
```

The second parameter of *BeginInvoke* specifies the delegate that is used as a callback function which will be called from a separate thread but the creation and handling of this thread is done by the system, not the programmer. This new thread will block on the *EndInvoke* call until the asynchronous operation finishes. The advantage of this model is that there is no need to be involved with instances of the Thread class and it

fits very well the case when a time-consuming request - or a request that may block - must be issued (good examples are read and write operations that block when there is no data available) without waiting for its completion in the calling thread. In fact it is very similar to other systems' facilities for issuing asynchronous requests, for example Symbian OS's active objects.

Multithreading and concurrent programming can become quite complex for the unexperienced programmer but they cannot be easily avoided in many cases, and managing communication between the computer and an RFID reader is one of the tasks where some kind of concurrent processing is definitely required, at least for performance reasons since the transfer of notifications and tag observations must be done as efficiently as possible, and as development goes on, a well-designed multithreaded system usually causes less headaches to the developers than some single-threaded, “traditional” design.

5.5.2 Network programming

In the demonstration system communication over a network occurs at least at two places: between the UHF reader and the computer that is running the server application, and between the server and the end-user application or the test programs. Even though at the final presentation everything ran on the same machine, this is not compulsory as the server program may have its own dedicated machine if needed but in our case it would have been an overkill since the amount of data that must be processed is definitely low so a middle-class PC can easily cope with the task of handling every software component of the system, the server, the database, and the presentation application. Nevertheless, the possibility of creating a system that is better distributed to more machines is there.

To implement the communication over the network, TCP sockets were used. The sockets interface, originating from the Berkley distribution of Unix, first appeared in 1983 and provides a way to implement communication between hosts connected through a computer network or between processes of the same machine. In the

demonstration system only Internet sockets using the reliable Transmission Control Protocol (TCP) were employed. Windows applications can use the Winsock programming interface that provides the Unix-style sockets functions. For more information of Winsock see *Jones, A. 1999.* and the *Winsock Programmer's FAQ* (http://tangentsoft.net/wskfaq).
In the .NET Framework most of these routines are encapsulated in the *Socket* class on the *System.Net* namespace. There are some extensions to make sockets programming more comfortable, for example, many methods have a corresponding *BeginXX/EndXX* method that use asynchronous delegate invocation. This can be very handy to implement behavior similar to non-blocking sockets without the inconvenience and little quirks of relying on the traditional non-blocking design.

The outline of the steps needed to implement a server that listens for incoming connections on a pre-defined port is the following: create a socket (create a new *Socket* instance), bind it to an address (call *Bind*, usually the address of the network adapter in the local machine is used) and a port, start listening (call *Listen*), and extract a pending connection from the waiting queue or block until a connection request arrives (call *Accept*). If a connection is established then data can be sent and received by using the *Socket* instance returned by *Accept*. Since in most cases multiple clients must continuously be served, a simple single-threaded design that blocks on every accept/read/write operation is not sufficient. There are three solutions:

The first is the case when after accepting a connection, a call to *fork* is made to clone the current process and create a new child process which will handle the communication to and from the newly connected client. This is only used in Unix applications since there is no direct equivalent of *fork* in Windows and, as mentioned earlier, creating processes and switching between them has a higher overhead than with threads. Also, communication between the processes can be difficult, because the only way is to use some inter-process communication facilities (like sockets or shared memory) for data exchange.

The second is to create a new thread for each connection. This is what is used in the server application (see Section 5.6) as it is relatively easy to implement and the number of simultaneous connections are expected to be low. Data sharing is easier in

this case, of course enough attention must be paid to synchronization. The disadvantage is the overhead when there are many simultaneous connections and thus many threads are created and lots of context switches occur.

The third solution is to use the *select* function and keep the program single-threaded. Alternatively this method may be mixed with multithreading in a way that each thread handles a few connections, not just one, so the number of threads needed is reduced. The *select* function checks the status of one or more sockets, this means checking for availability of data to read or a pending connection, writability (is there space in the underlying buffer?), or for error status. If no sockets from the specified list satisfy the given criteria (e.g. there is no data to read on any of them) then it will block until at least one of them comes into a status that satisfies the rules (e.g. data arrives) or until a specified amount of time has passed. By using this method the main processing takes place in a large loop that continuously calls *select* on all the client sockets (e.g. to check if there is data to read) and the server socket (to check if there is a pending connection request that could be accepted).

In the demonstration system the basic multithreaded approach is used but since the thread that handles a client has other tasks to do (e.g. checking if the exit flag, indicating that the system is shutting down, is set), blocking on a socket read operation is not acceptable and therefore the threads call the *Socket.Poll* method (that does the same as *Socket.*S*elect* but only checks one given socket) to check if there is data to read, if not then they block for a very short amount of time, do other operations, and start over with *Poll* again.

Clients that want to connect to a given server (i.e. the IP address and the port on which the server is listening is known) should create a socket and call *Connect*. After the connection has been established the same functions and methods are used for receiving and sending data as on the server side.

5.6 The server application

The server program was introduced early into the design of the system, at that point it was needed only because of one reason: to provide the ability of using the reader at the same time to multiple applications. Current RFID readers usually only allow one connection at a time, even when a TCP connection is used to connect to the device, and this can get quite uncomfortable when more than one developer is working actively on parts of the system that use the reader directly, especially when it comes to testing. Later, when more equipment was available, it was extended to provide seamless, transparent support for different readers.

The server application (internally called rfidserv) can be treated as something that is very similar to larger-scale middleware products. However, it would not be wise to call it a middleware since a very important feature of such products is deliberately not implemented: the processing and filtering of raw tag observations and converting them to events that are hopefully more meaningful to the higher-level applications. In rfidserv all the raw observations are forwarded to the clients, in a reader-independent fashion of course. So it can be thought as a multiplexer that also provides a level of transparency to hide the device-specific details. It handles network (TCP sockets) and serial (RS232) connections to readers and provides TCP connectivity to clients. The protocol used between the server and its clients is a line-oriented text protocol, modeled after the CHUMP of the SAMSys readers, see Section 4.1.2. All the data is sent in plain text by default but a simple encryption scheme using the RSA asymmetric algorithm as provided by the .NET Framework is also available. This is not meant to guarantee really secure data transfer though. To implement a secure and encrypted channel between rfidserv and the client applications SSL would be an ideal and not too complicated (regarding the time of implementation) choice.

For each reader a logical name is assigned, and the antennas of each reader are numbered from 1. More sophisticated naming of antennas (something like the concept of sources and readpoints in the *Reader Protocol Standard*, see Section 4.1) was decided to be unnecessary. Both the lack of any event processing and this relatively low-level device identification scheme were deliberate choices because the main goal

of the development was to test and research, any such features can be added at a later time, if needed. The same applies to the protocol used: ALE (see Section 4.2) could have been used but any testing application needs much more than just getting tag events, for example tag writing and changing reader settings (power levels, tag protocol settings, etc.) and there are no standards for how to handle these within the context of ALE or ALEPC. Of course, a real high-level application should not cope with such low-level hardware questions but in the research, development, and testing phase tools like rfidserv and the testing application can be very useful and are often unavoidable.

The protocol used between the server and the clients follows the traditional request-response model with the addition of multiplexing the asynchronous notifications also onto the same channel. Some of the requests (the "setter" ones that change some properties of the reader) have no response pair, i.e. no acknowledgment is sent back because TCP is a reliable transport protocol and the possibility of failing (e.g. due to a physical problem with the reader) is quite low. If really high reliability is required then the clients can still check if such a command succeeded or not by issuing a corresponding "getter" command. Requests to which a response will be generated must pass an arbitrary value besides the regular parameters. This value will also be included in the response and can be used for synchronization purposes even though it is guaranteed for clients that the responses will be sent to them in the order of the requests so the usage of this field can generally be omitted. An idea could be to use the value returned by *DateTime.Now.Ticks* but this may not be satisfactory because the uniqueness is not guaranteed at all since it depends on the resolution of the system timer which, according to the library reference, is approximately 10 milliseconds on NT-based systems.

After a client connects a short handshake is done during which the server sends the names and antenna information of available readers. The server will send the raw tag observations in the form of

```
T,<tag ID>,<tag type>,<antenna number>,<reader name>
```

terminated by LF.

Since these are the raw observation events without any smoothing and filtering, a tag that is in the field of the antenna will generate many such lines until it is moved out of

range. In an environment where many tags can stay in the field for a longer amount of time this would generate very high network traffic and would take increased processing power both on the side of rfidserv and its client applications. However in the demonstration system the load is acceptable without any further filtering and just getting the raw notifications as soon as possible is very handy for testing and measuring purposes.

Without detailed description the most important commands that the clients can issue are the following:

ignore_reader, accept_reader, ignore_antenna, accept_antenna	Enable/disable notifications about tags detected by the given reader or the specified antenna of the given reader.
get_supported_tagprotocols	Retrieve the list of tag protocols supported by the given reader.
get_scannedproto, scannedproto	Get/set the current tag protocol settings for the given reader, i.e. what type of tags will be "recognized" by the reader.
antenna_on, antenna_off, is_antenna_on	Turn on/off/query signal transmission on the given antenna of the given reader.
get_powerlevel, powerlevel	Get/set the transmit power level for the given antenna of the given reader.
get_gain, gain	Get/set the gain of the given antenna (if supported by the reader).
get_lbt, lbt	Turn Listen Before Talk on/off.
get_freqplan, freqplan	Some UHF readers, like the MP9320, may allow very sophisticated manual setting of the frequency hopping and frequencies that are used for transmitting.
read_user_data, write_user_data	Read/write the user data area of a HF tag.
write_id	Rewrite the ID on a rewritable UHF tag.
get_cap	Query the capabilities of the given reader (better said, the capabilities of the driver for the given reader).

Table 3. Some commands from the rfidserv protocol

Besides tag read notifications the server can also send notifications to clients about state changes of the readers. This means that when one client application makes a

change to some settings (e.g. issues a *powerlevel* command) all the other clients will be notified that the given setting has been changed so they can query the new value and update their behavior and user interface accordingly.

The rfidserv application was written in C# 2.0, targeting .NET Framework 2.0. While originally version 1.1 of the Framework was used, version 2.0 contains a set of classes for handling communication over the serial port and this (and other minor advantages, like enhancements in the language and the libraries) provided sufficient ground to justify the move to .NET 2.0. The program can be run in two modes: as a standalone console application and as a Windows service.

With the help of the libraries in the Framework the creation of services is not a complicated task anymore. When running as a service the server is nicely hidden in the background and can easily be started on Windows startup and it is only stopped when the machine is being shut down. Logging is done by writing both to the Windows event log and to a plain text log file. During development however problems and errors are discovered faster if messages are also shown in a console window, to go safe it was decided to stick with the method of running the server as a normal console application even at the exhibition. The end-user application runs in full-screen mode anyway, hiding anything else from the users (visitors).

The most struggling issue with the service mode was the problem of getting interrupted by the operating system during the system shutdown so connections to readers may not get terminated cleanly. This can cause very disturbing effects since if the connection to a SAMSys reader is not closed properly then the reader will think that the connection is active (even though the application and the machine that created the connection may already be dead) and will refuse any further connection request (since only one client can connect to the reader at a time). If this happens the only way to go is to power off and on the reader. Also the order in which the services in the system are terminated can cause surprising effects, for example if the event log service is stopped before rfidserv then subsequent log messages will be lost. One has to play with timeout and service dependency settings and test a lot to make sure that all of the shutdown activities are really executed instead of being aborted by Windows at some point.

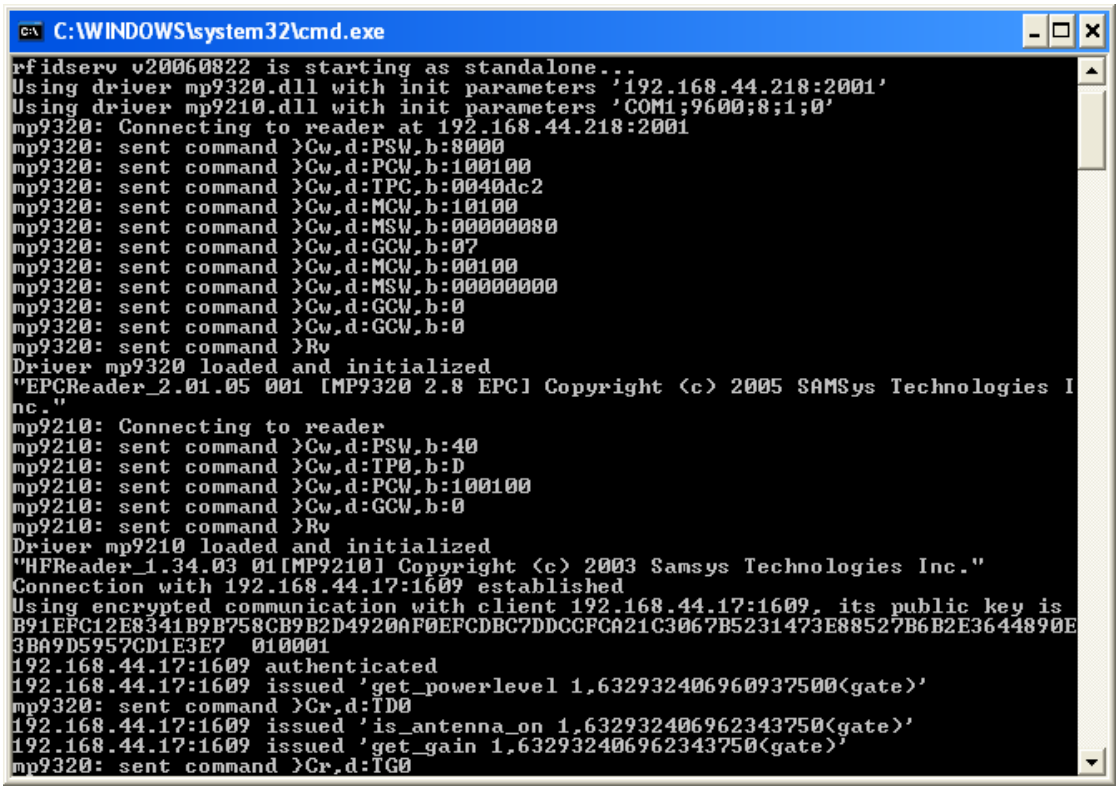

Figure 11. The log messages in the console window when running the server as a normal application. The CHUMP commands for reading and writing configuration registers are prominent.

While originally there was no need to support any other devices besides the MP9320, the server got modularized quite early. This means that the code responsible for maintaining the connection to the reader, processing of incoming data, and translating the requests issued by clients to device-specific commands was moved to a separate class that resides in a separate assembly (which becomes a DLL file in the filesystem) and was loaded dynamically by means of reflection facilities provided by the .NET Framework. At the end of the development process there were drivers for the SAMSys MP9320 and MP9210, the Phidget LF USB reader, and the Caen A948 (this one was preliminary and experimental, it was created to give proof that any reader that has a more or less decent API can easily be integrated into the system). For testing-without-device purposes a "dummy" driver was introduced: it does absolutely nothing but with the integrated tag simulator client applications can be made to think that a real tag was read.

A driver consists of one class that implements a pre-defined interface. When a reader-related command is received from a client, the corresponding method from the driver is called after a little pre-processing. The raw tag observations coming from the reader are usually simply forwarded to the higher-level layer by using the Framework standard event handling mechanism. The driver may use the vendor-provided API libraries for communicating with the reader or it may maintain the connection without any external libraries if the reader protocol is well-documented and for some reason the vendor-provided libraries cannot be used (e.g. they are not mature and stable enough) or do not exist.

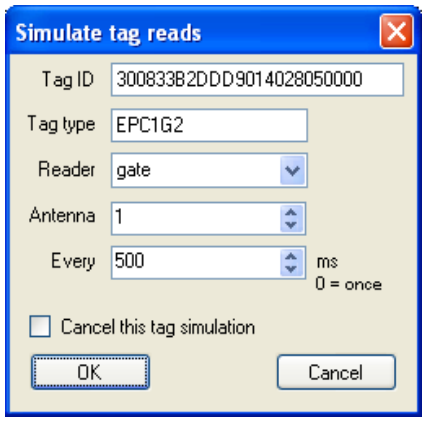

Figure 12. The tag simulator.

Besides the tag simulation setup dialog another dialog is offered for easy change of the readers that are present in the system and their settings (address and port in case of TCP and serial port settings in case of RS232).

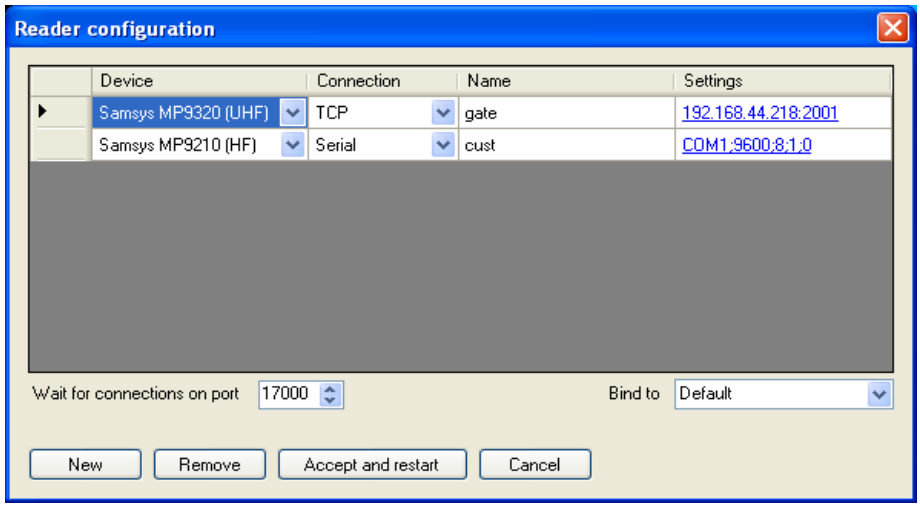

Figure 13. The reader configuration dialog.

The outline of the operation of the server application is the following:

```
Main thread:
1    Process command-line arguments.
2    Load the needed assemblies and create an instance for each
     "driver" class by using Assembly.LoadFrom and
     Activator.CreateInstance
3    Setup Control-C handler and keyboard input watcher thread.
4    Create Socket object, perform bind and listen.
5    Create and start the listener thread and either do a join (block
     until listener thread finishes, that is, the server is shutting
     down) or in case of running as a service the overridden OnStart
     method returns.

Listener thread:
1    while true do
2        issue a BeginAccept request
3        block until a pending connection request is extracted from
         the queue and the connection is established or a given
         ManualResetEvent object is set to signaled state
4        if exit flag is set then break
5    close server socket
```

The blocking accept cannot be used since if for example a server shutdown or restart is requested then all the threads should exit cleanly as soon as possible. The callback passed to *BeginAccept* will create and start a new client handler thread. If one studies the steps of the listener thread described above and the asynchronous delegate invocation example in Section 5.5.1 carefully then an interesting question may arise: how can an asynchronous request be canceled? It is not enough if the listener thread terminates cleanly when the event is signaled and the exit flag is found to be set, the thread on which the callback of *BeginAccept* executes will still be alive, blocking on *EndAccept*. Strangely enough, it seems that there is no support for canceling a request once it has been issued. Fortunately the case of accept carries the (decent though not really nice) workaround in itself: when the listener thread closes the server socket *EndAccept* will return.

The client handler thread polls the client socket in a loop for data to read with a timeout of one second (that is, *Poll* will block at most for second if there is no data available). If data arrives then it is appended to the buffer and if a full line (some text terminated with LF) is available in the buffer then it is extracted and processed. If the remote host closes the connection or a fatal socket error occurs then the thread exits. The timeout is needed since the check for server shutdown must be done periodically, even if the client issues no requests.

When running as a standalone console application the keyboard input watcher thread is responsible for watching if a key is pressed. Not surprisingly a simple *Console.ReadLine* call would not be sufficient since it would block the thread until enter had been pressed. Fortunately *KeyAvailable* and *ReadKey* are provided in .NET 2.0. If the user issues the exit command on the console then the exit flag is set and the event, on which the listener thread is waiting, will be signaled.

When running as a service there is no console available but the user must still be able to execute at least the configuration or the tag simulator dialog. For this a notify icon is provided on the system tray which brings up a menu when the user right-clicks it. Needless to say, this involves another thread which is responsible solely for calling *Application.DoEvents* in a loop to get the the normal Windows events processed and so to keep the icon and the menu working.

Even though every software component of the demonstration system was written in C# and thus is targeting the .NET Framework, clients for rfidserv are not limited to the .NET environment: any application running under any operating system can connect if there is a working network connection and TCP sockets are available. For the test and end-user applications a connector library was created which can also serve as a reference implementation for creating and maintaining connections to an rfidserv instance. It encapsulates every command from the communication protocol described above. The following piece of code is probably the most simple example, it just dumps the information on the output when a tag read event arrives.

```
RfidservConn c;
void tag(object sender, TagReadEventArgs e)
{
    Console.WriteLine(DateTime.Now + ": " + e.TagId
        + "  " + e.TagType + "  " + e.Antenna + "  "
        + e.ReaderId);
}
...
    using (c = new RfidservConn(Dns.GetHostEntry(
        Dns.GetHostName()).AddressList[0].ToString(),
        17000, true))
    {
        RfidReader r = c.GetReader("gate");
        r.AntennaOn(1);
        // set transmit power level for antenna 1 to 50%
        r.SetPowerLevel(1, 50);
        c.OnTagRead += tag;
        c.StartTagListen();
        Console.ReadLine();
    }
```

In the background a TCP socket connection is maintained to the server by using separate threads. To prevent performance and deadlock problems the event handler (the *tag* method in the example) is called from a dedicated thread that does nothing but "fires" the events that are placed into a synchronized queue by another thread, the thread that is responsible for handling the incoming data from the server.

5.7 The end-user application

The end-user application (also referred to as the presentation application) consists of three independent modules: the logic that implements the behavior described in the use case, the user interface, and the database support. The separation of the logic from the UI provides the possibility of easy reuse and fast switch to a different presentation layer. For example, the application (called GateDemo internally) had no graphical interface at all in its first version, just some information was dumped into a console window about the state of system, and still, the sources for the logic implementation

remained completely unchanged even after introducing the GUI. The logic informs the user interface about the system state changes by using the standard event handling mechanism of the .NET Framework.

The GateDemo application was written in C# 2.0, using Visual C# 2005 Express Edition, and thus is targeting version 2.0 of the .NET Framework. Its main goal is simple: handle the tag observation events coming from the reader indirectly through the server by checking if the ID is present in the database and if yes, it shows name and price information on the screen in form of a list. While every tag used in the system has a different ID, they are not always attached to different products. That is, if there are two bottles of the same kind of beer then there will be two entries in the tag database because their tags will hold different IDs, but there will be only one entry for name, price, and other product details. In case of a relational database this means that one table holds the product names, prices, and possibly other details, another contains the tag IDs, and the two tables are in a one-to-many relationship using a foreign key reference.

As a deliberate design decision no user interactivity was involved, e.g. instead of requiring the customer to click a button after walking through the gate, two timeout periods were defined after which the system goes into a new state if the conditions are met. One is obvious: when the customer does not get identified for a short period of time the purchase is canceled, the transaction is rolled back. The other timeout is used to determine when the shopping cart is considered to have moved out from the field of the gate. It is not so trivial to come up with a solution that suits the need and behavior of the majority of people. In this system the process of item discovery is considered to have ended when no new items are discovered for a pre-defined amount of time (some seconds). Naturally it is very important to tune the amount of this time interval according to customer needs, e.g. a short timeout may cause confusion when someone is passing through the gate too slow, but if the value is too big then the amount of idleness can make the customers impatient and nervous.

The diagram at the end of this section presents the logic that defines the behavior of the GateDemo application. A photo of the demonstration system display showing the finished application, as it was presented to the public, can be found in Appendix 1.

The database server was MySQL 5.0 and the database for the system consisted of five tables and some stored procedures. As it can be seen on the flowchart, a database query must be issued when a tag is read by the UHF reader. To prevent the performance loss caused by passing many SQL SELECT statements repeatedly to the database a simple caching mechanism based on a hash table was used. Out of the five tables only three are compulsory for the demonstration system, the other two are needed for logging the completed purchase transactions.

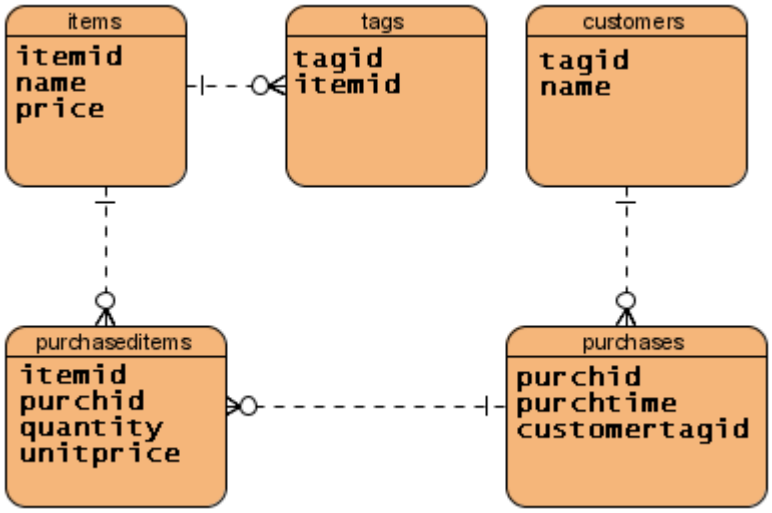

Figure 14. The database schema

The user interface is quite simple, with border-less windows and a little bit of transparency. The main control, showing name, quantity, and price information, is a DataGridView component. Since the logic is running on its own thread the problem of requesting user interface updates from a thread other than the main thread is present. To solve this the logic never calls an event handler directly. Instead, the events are queued and the main thread must periodically call a method that checks this queue and fires the events, if there are any.

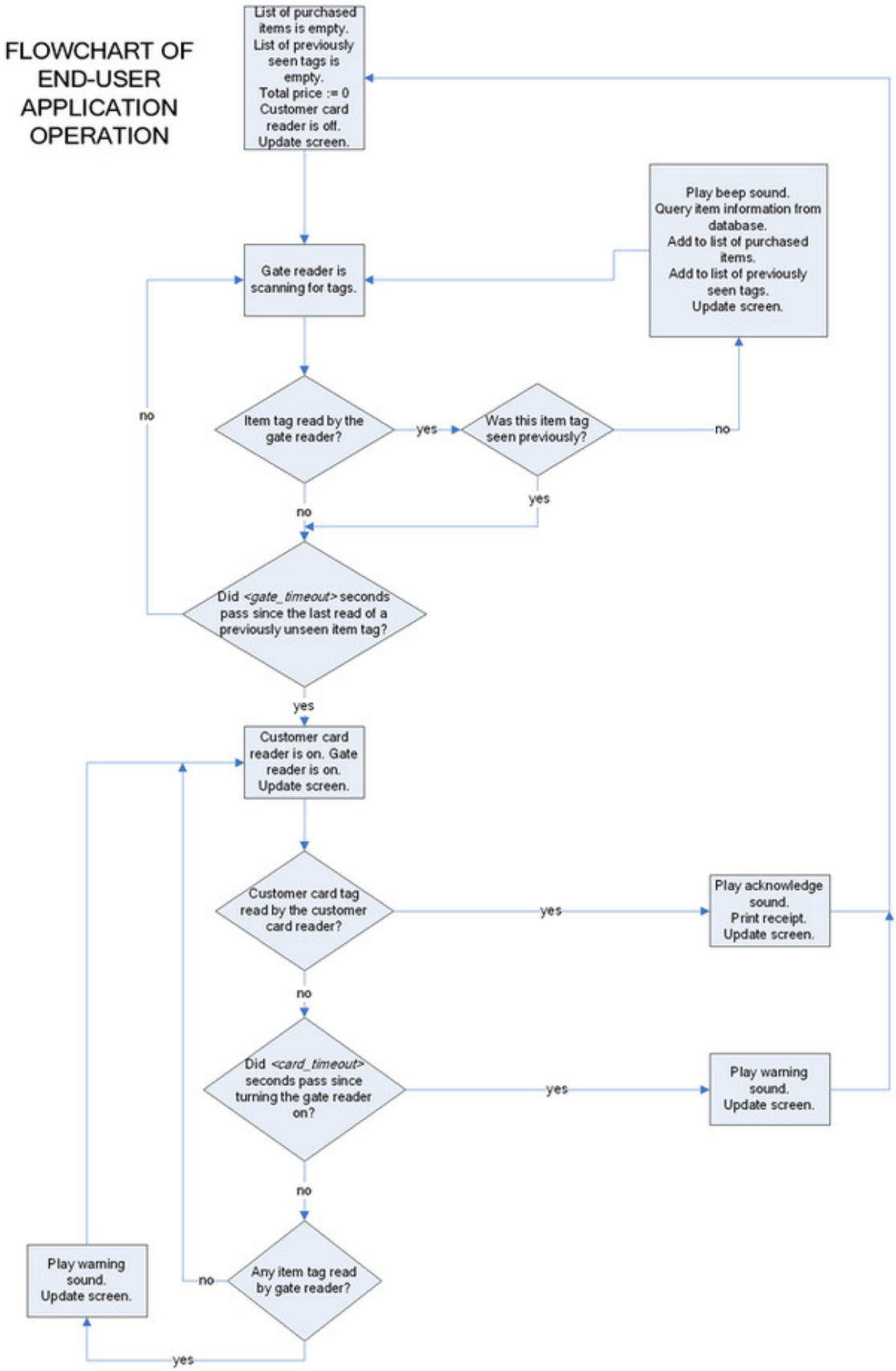

Figure 15. A flowchart that describes how the end-user application should operate, matches the use case in Section 5.2.

6 RFID NOW AND IN THE NEAR FUTURE

RFID is undisputedly successful and is becoming more and more beneficial at factories, in the supply chain, in logistics, but what real benefits does it give to the regular people, who do not care if the box in which the item, that he or she has just taken down from the shelf of the supermarket, arrived at the gates of the shop's depot was tagged or not? Working installations of automatic motorway tolling systems and libraries providing book check-out and check-in without staff interaction can be found throughout the world, to name two examples. However, the spreading of RFID technologies in retail at the consumer level seems to be a bit slow when compared to these which is quite unfortunate since an automatized shopping experience would be a spectacular proof for the capabilities and matureness of RFID and could revolutionize shopping habits and the way the consumers think about shopping.

In the near future reliability and reading distances are likely to increase, especially since active tags will get more prominent. A solution to their power source requirements may be the printable paper batteries, and such technologies are already under constant research and development. The vision is that active tags should become similar to the current sticker-type passive ones with a long-enough life and low-enough manufacturing cost. Meanwhile, the size of both passive and active tags should decrease: while today's sticker tags are considerably smaller than their ancestors, they are still not small enough for being attached to every product. How would one place a regular sticker tag on a pack of chewing gum for instance?

Not even a successful RFID installation in itself can guarantee any real groundbreaks, it must be understood that RFID is just a component, the success of the whole system depends heavily on other parts and on how well they integrate with each other. For example, our shopping demonstration was focused on the problem of recognizing what items the customer has in the shopping cart. In a real shop environment lots of other questions must be answered and solved, like how the payment is carried out or how seamless the co-operation regarding tagging is in the entire supply chain.

There are designs and prototypes for such integrated systems, see for example the MyGROCER (http://www.eltrun.gr/mygrocer/default.asp) project that “aims to exploit the opportunities that emerging mobile telecommunication and electronic commerce technologies and automatic product identification technologies offer to the retail sector”. Its aim is more than just providing a self-checkout experience at the supermarket, that is just one component. It aims to get RFID and the network infrastructure not just into the store but also into the consumer's home. As technology is developing in a fast pace, the vision of, for example, intelligent refrigerators that know what is inside them, when those products expire, etc. does not seem to feel as weird as it felt a few years ago.

SUMMARY

As described in Section 5.4, setting up readers and antennas is very often a tedious, trial and error task with a lot of experimentation. The more the placement and orientation of tags can vary the more difficult it is to set up a system that guarantees a missed tag ratio close to zero. The vendor-provided software and documentation is often immature and does not give access to the reader's full potential. While this is likely to change in the near future, at the moment getting started with a reader that just arrived from the manufacturer or the reseller can be more complicated than it should be. Middleware products are usually aimed at enterprise-level systems and not small-scale environments, as it is often realized after checking out requirements, licensing costs, and the complexity of setup. Building up an own software environment from scratch needs experienced programmers, while hobby users at home may get along with using RFID technologies for simple and creative entertainment use (see for example *Graafstra, A. 2006.*); implementing a reliable RFID subsystem that integrates well with other components of the system requires dedicated engineers and programmers backed with lots of learning and research.

Our shopping demonstration as presented at the Tekniikkia 2006 exhibition (see *Appendix 1*) was considered a success. While many have some ideas about what RFID is capable of and what benefits it may provide to everyday's life - like finishing shopping by walking through a gate without spending time in a queue, having bar codes read one by one, and counting cash while other people are waiting in the queue more and more impatiently - a large number of visitors were still struck and surprised that the system really works and the computer magically knows what is inside the shopping cart. This is believed to be caused by the fact that the number of similar systems that are deployed (e.g. installed at the local supermarket) and ready for production use (i.e. consumers get in touch with them day by day and have positive experiences) is surprisingly low at the time of writing.

REFERENCES

Glover, B., Bhatt H. 2006. RFID Essentials. O'Reilly.

Kleist, R., Chapman, T., Sakai, D., Jarvis, B. 2005. RFID Labeling. 2nd ed. Printronix, Inc.

Graafstra, A. 2006. RFID toys: cool projects for home, office, and entertainment. Wiley Publishing.

Class 1 Generation 2 UHF Air Interface Protocol Standard Version 1.0.9. 2005. EPCglobal.

Reader Protocol Standard, Ratified Standard, Version 1.1. 2006. EPCglobal.

Application Level Events (ALE) Standard, Ratified Standard, Version 1.0. 2005. EPCglobal.

EPC Tag Data Standard, Ratified Specification, Version 1.3. 2006. EPCglobal.

ETSI EN 302 208-1 v1.1.2 European Standard. 2006.

ETSI EN 300 220-1 v1.3.1. European Standard. Sept. 2000.

Rao, K.V.S., Nikitin, P.V., Lam, S.F. 2005. Antenna Design for UHF RFID Tags: A Review and a Practical Application. IEEE Transactions on antennas and propagation, Vol. 53, No. 12, December 2005.

Eeden, H. 2004. Impact of Proposed Spectrum Regulations for Europe on UFH RIFD applications. White Paper. iPico Identification (Pty) Ltd, South Africa.

Priehs, E., Talbot, S. 2004. Radio Frequency Identification (RFID) As a Fixed Asset Management Solution. White Paper. Asset Management Resources, USA.

BITKOM RFID Project Group 2005. RFID White Paper - Technology, Systems, and Applications. BITKOM German Association for Information Technology, Telecommunications and New Media e.V.

Matrics, Inc. 2004. EPC and Radio Frequency Identification (RFID) Standards. White Paper. Matrics, Inc., USA.

[TI2005-1] Texas Instruments 2005. EPC Primer. 8th Texas Instruments Developer Conference (TIDC) 2005 India

[TI2005-2] Texas Instruments 2005. UHF Gen 2 System Overview. 8th Texas Instruments Developer Conference (TIDC) 2005 India

[TI2005-3] Texas Instruments 2005. UHF Applications - Installation Hints. 8th Texas Instruments Developer Conference (TIDC) 2005 India

Brock, D.L. 2003. The Electronic Product Code as a Meta Code. AUTO-ID Center White Paper.

MP9320 V2.8e UHF Long Range Reader User's Guide. 2005. First Edition. SAMSys Technologies, Inc., Canada

[CHUMP2005] Comprehensive Heuristic Unified Messaging Protocol Reference Guide for UHF Readers. 2005. Sixth Edition. SAMSys Technologies, Inc., Canada

MP9210 13.56 MHz Proximity Reader User's Guide. 2003. Second Edition. SAMSys Technologies, Inc., Canada

Technical Information Manual: CAEN UHF RFID Readers Communication Protocol. Revision n. 0, 27 October 2005. C.A.E.N. S.p.A. Italy.

Technical Information Manual: UHF Long Range RFID Reader Mod. A928. Revision n. 13, 3 November 2005. C.A.E.N. S.p.A. Italy.

Vogt, H. 2002. Efficient Object Identification with Passive RFID Tags. Department of Computer Science, Swiss Federal Institute of Technology

Sarma, S., Weis, S., Engels, D. RFID Systems and Security and Privacy Implications. Auto-ID Center, Massachusetts Institue of Technology.

Jones, A., Ohlund, J. 1999. Network Programming for Microsoft Windows. Microsoft Press.

Network Programming. .NET Framework Developer's Guide. MSDN Library. http://msdn.microsoft.com/library/

Managed Threading. .NET Framework Developer's Guide. MSDN Library. http://msdn.microsoft.com/library/

Finnish Libraries' RFID Working Group. 2005. RFID Data Model for Libraries: Finish Data Model. Final document.

Toshiro, M. 2005. Needs and Benefits of Massively Multi Book Agent Systems for u-Libraries. Massively Multi-Agent Systems I p. 239-253. Lecture Notes in Computer Science, Springer Berlin.

APPENDICES

Appendix 1: Presentation of the system at Tekniikka 2006

The demonstration system was presented to the public on behalf of the Jyväskylä University of Applied Sciences at the Tekniikka 2006 exhibition that was held 4-6 October 2006 in Jyväskylä. The system consisted of the two SAMSys readers, one circularly polarized UHF antenna, a printer, and a computer which was shared with another project and had dual display. One of the monitors was used for showing the end-user application while the other was used for the purposes of the other project.

Figure 16. The antenna as it was mounted at the exhibition.

Fortunately it was possible to mount the antenna at a relatively hidden location. Also as it can be seen no real gate was constructed at the exhibition site due to the fact that the one single antenna could be mounted at a suitable place without any other

components. The metal foil (covered in paper for aesthetic purposes) was used to limit the spreading of the signal so that the tags on the items in the shopping cart waiting for gate walk-through did not get read too early.

Figure 17. The shopping cart used at the exhibition with the paper backing and some tagged items.

The shopping cart was the same one that had been used in the laboratory (described in Section 5.4). The cardboard backing was also there, covered in paper for a better look. The items were randomly selected grocery products with the deliberate omission of liquid material. The recognition ratio was considered acceptable, with occasional missed items. Some of these misses were caused by tags touching or being too close to the metal sides, these were considered expected faults, while the reason for others was either the wrong angle of the tag or the material of some other product that hid and shielded the item's tag.

Figure 18. The display with the presentation application, the HF reader used to detect customer cards, and the printer used to print receipts.

Instead of just giving the plain plastic card format tag to visitors the card was attached to the back of a regular mobile phone. The idea was to give a hint for a possible future extension and implementation of the actual payment process. It could be done with a Near Field Communication-based system for example, which uses the customer's NFC-enabled phones for payment instead of any additional credit or other kind of cards. The main resource for more information about NFC is the homepage of the NFC Forum (http://www.nfc-forum.org).

Appendix 2: Plans of a simple RFID library system

During the development of the shopping cart-based demonstration system we had a short-lived side-project, the goal of which was to design and study how our, or other similar widely-available and relatively inexpensive equipment, could be used to implement a simple, small-scale library system, concentrating on the automatic, self-service check-out and check-in operations. The system would consist mainly of a gate that is only used for security and another reader or at least a dedicated antenna for the self-checkout/checkin desk, combined with a touchscreen device. The sub-project was abandoned after a short design period because the shopping system was based on UHF technology and the needs of a library environment seem to be better served by devices and tools using HF RFID, see Section 3.3. The following two diagrams show how the two most important modules of the demonstration software would have operated.

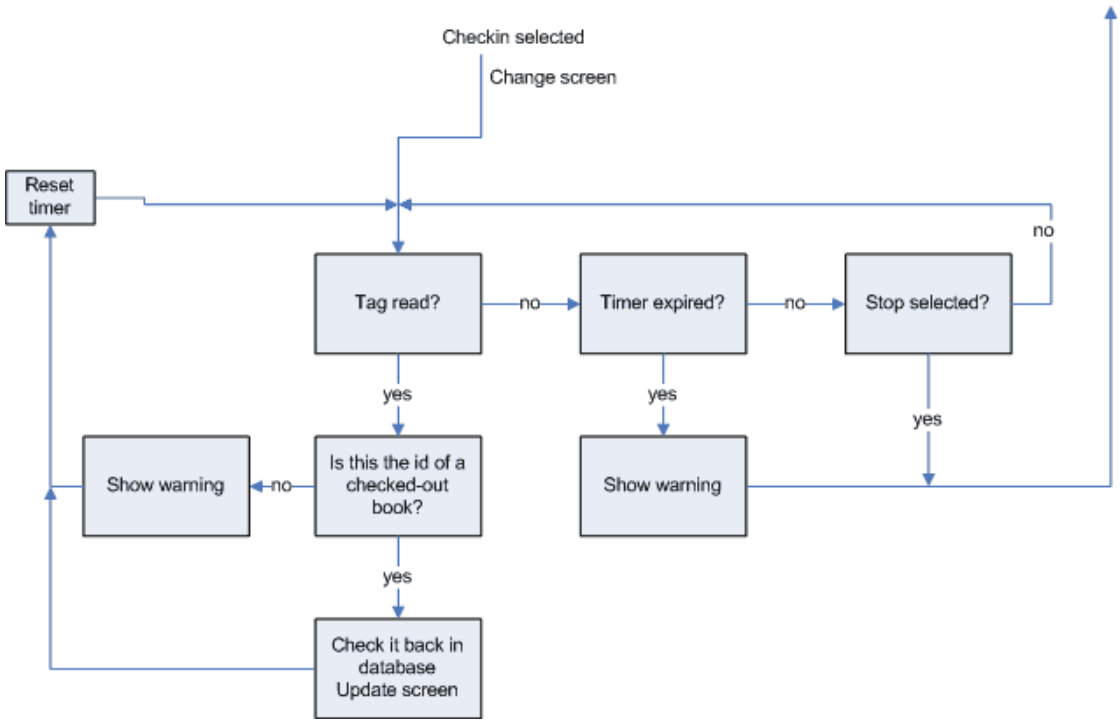

Figure 19. Flowchart for the implementation of the check-in (bringing back one or more loaned books) operation.

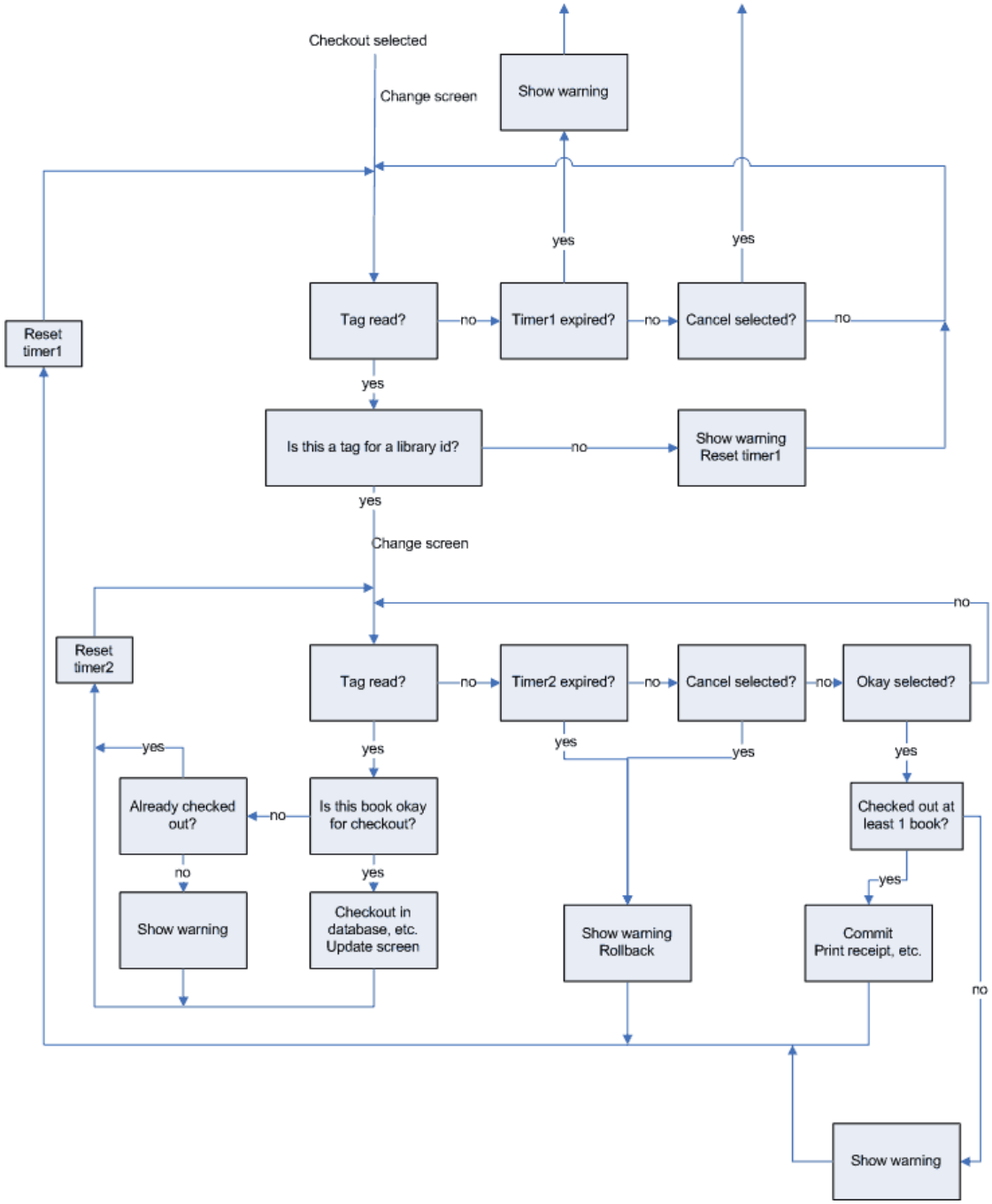

Figure 20. Flowchart for the implementation of the check-out (loaning books) operation.

Appendix 3: List of abbreviations

ALE	Application Level Events
ALEPC	ALE extensions for tag writing.
CR	Carrige Return
CRC	Cyclic-Redundancy Check
EIRP	Effective Isotropic Radiated Power
EPC	Electronic Product Code
ERP	Effective Radiated Power
ETSI	European Telecommunications Standards Institute
GUI	Graphical User Interface
HF	High-Frequency
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
LF	Low-Frequency or Line Feed
RFID	Radio Frequency Identification
RS-232	A standard for serial data interconnection. RS stands for RETMA Standard.
RSA	An algorithm for public-key encryption. The name comes from the initials of Rivest, Shamir, Adleman.
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UHF	Ultra-High Frequency
XML	Extensible Markup Language