Local Control in a Distributed Automated Identification Environment

by

James Luke Waldrop III

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2001

© James Luke Waldrop III, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author

Department of Electrical Engineering and Computer Science

February 4, 2001

Certified by

Sanjay E. Sarma
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Local Control in a Distributed Automated Identification Environment

by

James Luke Waldrop III

Submitted to the Department of Electrical Engineering and Computer Science on February 4, 2001, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Auto-ID center is investigating technologies that enable integrated applications concerning home automation, inventory control, information access, and physical tracking of every physical object. These technologies seek to catalogue all items in disparate databases and network many common household appliances (among other things) to utilize the data stored in these manufacturer databases. In addition, Auto-ID is prototyping local controllers needed to manipulate or filter information as it is collected from various sources, and perform localized control decisions to take load off of the overall system. This thesis gives a brief overview of all technologies included in the Auto-ID project and details several prototypes for local network and Auto-ID tag reader control systems.

Thesis Supervisor: Sanjay E. Sarma
Title: Associate Professor
Acknowledgments

The author wishes to thank:

• Prof. Sanjay Sarma, for taking me on as a student.

• Dr. Daniel Engels, for overseeing the production of this thesis.

• Joseph Foley and Deborah Moore, for working in the same office as me for several months, including the time spent writing this thesis.

• My family, for educating me before I got to MIT.

• Jennifer Clay, for tolerating me while I did the following work.
## Contents

1 Introduction .......................... 9
   1.1 Overall problem ..................... 9
   1.2 Main Components of the Auto-ID system ....... 10
   1.3 Savant Problem Specification ............ 11
   1.4 Related Work .......................... 11
   1.5 Overview of Thesis .................... 13

2 Overview of the Auto-ID networking system ....... 14
   2.1 Vision .................................. 14
      2.1.1 Ideal Auto-ID system .............. 14
      2.1.2 Engineering Restrictions .......... 15
      2.1.3 Required Network Characteristics .... 15
   2.2 Overview of Auto-ID Architecture ........ 17
      2.2.1 ePC overview ...................... 17
      2.2.2 ONS overview ...................... 19
      2.2.3 PML overview ...................... 20
      2.2.4 Savant overview ................... 21
      2.2.5 XQL queries in PML requests ...... 22
      2.2.6 RFID Tags .......................... 22
   2.3 Current Local Network Paradigm ........... 23
      2.3.1 Query/Response System .......... 23
      2.3.2 Mechanism .......................... 25
      2.3.3 Savant and Workshops .............. 26
3 Savant Specification

3.1 General Classes of Data Queries

3.1.1 Virtual Database Queries

3.1.2 Physical-space Database Queries - Physical Databases

3.1.3 Dynamic Database Requests - Mechanisms, Savants and Idiot-Savants

3.2 Characteristics of an ideal idiot-savant

3.3 Characteristics of an ideal savant

4 Savant Implementations

4.1 Overview of Component Systems

4.1.1 MySQL

4.1.2 LonWorks

4.1.3 Perl Modules

4.2 Enhanced CD Player

4.2.1 MySQL Database Table Setup

4.2.2 cd-input.pl

4.2.3 scan-CD.pl

4.2.4 get-playlist.pl

4.2.5 punt-playlist.pl

4.2.6 playlist-CD.pl

4.2.7 CGI script: http://omaha.mit.edu/cgi-bin/PML

4.2.8 Enhanced CD Conclusions

4.3 Intelligent Shelves with LonWorks Readers

4.3.1 MySQL Database Table Setup

4.3.2 Virtual Tag Reader Network - BBox/randomize.pl

4.3.3 timestamp.pl

4.3.4 test-get.pl

4.3.5 CGI Script: Savant
5 Conclusions

A Enhanced CD Code

A.1 Code: cd-input.pl ........................................... 54
A.2 Code: get-playlist.pl ...................................... 55
A.3 Code: punt-playlist.pl ...................................... 56
A.4 Code: scan-CD.pl ............................................. 57
A.5 Code: playlist-CD.pl ......................................... 60
A.6 Code: wipe-db.pl ............................................. 61
A.7 Data: sample CD data file ................................. 62
A.8 CGI code: http://omaha.mit.edu/cgi-bin/PML ............... 62

B Intelligent Shelves Code

B.1 Code: Virtual Tag Reader Network - BBox/randomize.pl ....... 64
B.2 Code: timestamp.pl .......................................... 65
B.3 Code: test-get.pl ............................................. 68
B.4 CGI Code: Savant ............................................. 68
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>An overview of where various subsystems fit in the Auto-ID system</td>
<td>18</td>
</tr>
<tr>
<td>2-2</td>
<td>The 7 types of device in our system</td>
<td>24</td>
</tr>
<tr>
<td>2-3</td>
<td>The path of a full ONS/PML query</td>
<td>26</td>
</tr>
<tr>
<td>4-1</td>
<td>An overview of the Enhanced CD Player</td>
<td>40</td>
</tr>
<tr>
<td>4-2</td>
<td>An overview of the Intelligent Shelves savant</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

4.1 LonWorks Theoretical Maximums ........................................ 38
4.2 LonWorks Transport Media Bandwidth .................................. 38
Chapter 1

Introduction

1.1 Overall problem

The Auto-ID center's main thrust is to place on every object an inexpensive short-range RFID tag with an on-chip unique identifier called an ePC (Electronic Product Code). The ePC supports one unique identifier per item, allowing the tracking of individual objects. This causes a massive increase in complexity. Corollary to the existence of ePC tags is the ability to support the surveillance of ePC tags in a myriad of environments - in warehouses, on shelves in a supermarket, and in a domestic environment. In addition, the Auto-ID center aims to make dynamic data available in a secure fashion through a distributed server system on every individual item tagged with an ePC.

The overall effect the Auto-ID center wishes to produce is the storage of all information vital to an object – any object, not just those that are UPC coded today – on that object, in the form of an Auto-ID tag. In the Auto-ID system, objects can communicate amongst each other and with items that may be able to utilize the information stored on an object.

The vision of the Auto-ID center is to have a system where all objects can communicate with one another any information one object might need on another. A supermarket shelf could determine when the milk stored upon it has expired. A package of pasta could communicate with a microwave in which it had just been placed and give
instructions on how to cook the pasta. Shipping, inventory and warehousing would be revolutionized as every item could be tracked in real time, eliminating bottlenecks and supply gluts. An ideal, but not feasible, solution to the vision presented above is to provide all objects with all relevant information about that object, and a means of retrieving any information they need from any other object. Objects would be able to communicate with any other object that wished to communicate with it, in complete informational security. However, with current technology, the sheer amount of data required to cover every possible inquiry cannot be stored on an object itself, and updating this data dynamically would also be technically infeasable. However, while we cannot store all possible data about an object on the object itself, we can tag the object with a unique identifier and store all relevant information about the object in a massive distributed database system. This, then, is the route the Auto-ID system is taking.

1.2 Main Components of the Auto-ID system

To produce the effect described above, the Auto-ID system is divided into several parts:

**Tag** Every object in the Auto-ID system has a small, inexpensive passive radio-frequency identification tag embedded somewhere within it that can be “read” by an RFID tag reader. Each tag’s unique identifier is an ePC.

**ePC** The ePC is a unique identifier embedded within each RFID tag, and additional devices may have their own ePC codes that are never embedded within an RFID tag. This ePC is analogous to an IP address. The ePC code of an object allows the ONS system to find where information on that object is stored.

**ONS** The Object Name Service, ONS, is a distributed system built on top of DNS to provide a means of easily and quickly converting the unique ePC inside each tag into a pointer to a PML server with data on that object.
PML

Product Markup Language, or PML, is a derivative of XML that will be used by the Auto-ID project to markup and compile pertinent data on all objects covered by the Auto-ID system.

Savant

The savant is the local control system. It interfaces with objects and retrieves information about them, using ONS to find the PML server. It can also interact with other savants, and caches a good deal of the object information locally to cut down on bandwidth utilization. While the four disparate components above constitute a larger system, the Savant links them all together.

1.3 Savant Problem Specification

This thesis concerns prototype “local” systems that acquire a unique identifier from an ePC tag and must retrieve information from ONS and PML systems without becoming a bandwidth hog. The local systems must cache information and make intelligent requests, coupling their PML queries with XQL filters. Systems may also need to make information available to the outside world, whether passively waiting for a data query or actively sending data to other systems. The massively distributed nature of the Auto-ID system must be taken into account in these cases as well. Detailed in the following chapters are suggestions for ideal savant systems and descriptions of savant implementations that I have already implemented.

1.4 Related Work

LonWorks

The Auto-ID center is investigating the use of LonWorks, by Echelon Corporation, as its local networking protocol. Several (approximately 250) manufacturers have joined together in the LonMark coalition and have produced products capable of communicating over a LonWorks network with the LonTalk protocol, and given the Auto-ID goal of standardized high-information communications between devices and objects, LonWorks is a natural choice.
The Neuron Chip is the heart of most commercially available LonWorks devices. In essence, it is an 8-bit microprocessor with the LonWorks protocol hardcoded onto it, programmable in Neuron C (a superset of ANSI C). The Neuron Chip implements the bottom 6 layers of the OSI 7 layer paradigm for LonWorks in firmware, leaving the application layer to the LonWorks developer. In addition, the LonTalk protocol has very flexible addressing protocols that are well suited for the Auto-ID project’s needs for data filtering and bandwidth minimization.

**Passive Motorola RFID tags** Motorola Corporation has been supplying the Auto-ID project with passive low-price RFID tags capable of holding 40 and 96 bits of information, as well as RFID tag readers for these tags. The Auto-ID center has decided upon passive tags (which wait to be polled for data) as opposed to active tags (which actively send data to any readers within range) due to price concerns. In addition, passive tags can be much smaller than active tags, since many passive tags do not require a power source. However, a passive tag can store much less data than an active tag, and anti-collision algorithms (to allow multiple tags to be read by one reader) are only coming out of prototype stage now. While the tag price currently is on the order of $0.25 per tag, Motorola is working to reduce this price, and is shipping the Auto-ID project LonTalk-capable RFID tag readers to interface with our local network choice more readily.

**MySQL** MySQL is a relational database management system, or RDBMS, distributed under a very liberal Open Source copyright. Due to its open source nature, it was a natural choice for a massive system that could not support a more expensive DBMS like Oracle due to the sheer number of databases that would need to be maintained. MySQL, unfortunately, does not currently have the capacity to output information in an XML format or to accept an XQL query directly. However, the field of XML query languages is a new one, and using public-domain Perl modules
we have been able to code a “shell” around the MySQL DBMS systems used in the savant prototypes detailed in this thesis that can circumvent this shortcoming in the current versions of MySQL.

1.5 Overview of Thesis

In Chapter 2 I go over the main technologies that the Auto-ID center is using to realize its vision, and some of the reasons for these choices. I also introduce the Savant paradigm, which is briefly discussed in Chapter 2 and discussed in detail in Chapter 3. In Chapter 3 I explain the applications that a savant would be used for, some of the workings of a savant, and how an “ideal” savant would be constructed. Then, in Chapter 4, I provide two prototype savants and explanations on how they were constructed, why they were constructed in that way, and how well they turned out. I conclude with features that could have improved the prototype savants and suggestions for engineers implementing savants for the Auto-ID system in the future.
Chapter 2

Overview of the Auto-ID networking system

2.1 Vision

The vision of the Auto-ID center [4] is to have a system where all objects can communicate with one another any information one object might need on another. A supermarket shelf could determine when the milk stored upon it has expired. A soda can could inform a robotic arm attempting to pick it up how to do so without crushing the can. A refrigerator could monitor the food products most commonly entering and leaving cold storage and could compile a suggested shopping list for its owners, which would be combined with the similar list produced by the kitchen cabinet and the toiletries shelves in the bathroom. Shipping, inventory and warehousing would be revolutionized as every item could be tracked in realtime, eliminating bottlenecks and supply gluts.

2.1.1 Ideal Auto-ID system

An ideal solution to the vision presented above is to provide all objects with all relevant information about that object, and a means of retrieving any information they need from any other object, with a small, cheap data storage, retrieval, and
processing unit embedded within the object. This hypothetical unit would be able to communicate with any other object that wished to upload or download data to it, without restrictions of range, bandwidth, or information density. All informational transactions would be absolutely secure. No one should be able to know that, for example, a wealthy doctor purchased a diamond necklace that did not end up in his wife’s possession.

2.1.2 Engineering Restrictions

The ideal solution presented above is not possible with current technology. In addition, it is reasonable to assume that it will not be possible in the near future, as the amount of data that people will wish to transfer between objects will grow just as the storage and bandwidth capabilities of embedded systems grow. However, it is possible with judicious distribution of processing power to make a reasonable approximation to the above vision. In short, while we cannot store all possible data about an object on the object itself, we can tag the object with a unique identifier and store all relevant information about the object in a massive distributed database system.

The Auto-ID system revolves around a passive radio-frequency identification tag. This tag must be extremely inexpensive - pennies per tag - as it will become more widely used than the ubiquitous UPC barcode. With this in mind, the Auto-ID center has settled on a passive tagging system holding a mere 96 bits of data - enough to give every object a unique identifier, but not enough to store much pertinent information on the object. Instead, the information on the tagged object will be stored in an Internet-capable database. The general idea, therefore, is to read this unique ePC identifier from an object and then retrieve its data from the manufacturer’s PML server (see section 2.2).

2.1.3 Required Network Characteristics

The overall project is daunting. As we said above, we cannot store all relevant data on all relevant objects on the objects themselves, but we can barely do so on a
massive distributed network - the Internet. To produce a system capable of serving unique data about every object on earth, the system must be designed with absolute efficiency and massive distribution in mind.

**High bandwidth** The number of items that will need to be tracked by this system is staggering. Proctor and Gamble reports that they sell hundreds of millions of boxes of Tide detergent every year. Gillette Corporation reports that they sell over a billion razor blades every year. Each of these items will need to be tagged. Information will need to be maintained for each item. While most information will be redundant and easily compressible, a huge number of requests for information will be incoming to any machine capable of serving this data at any time. Each response will be a PML file (see section 2.2) possibly on the order of megabytes in size.

**Data filtering** We take two steps to deal with the huge load that will be put on our Auto-ID information retrieval service. The first step is a judicious use of data filtering. We can use query parameters in PML queries (see section 2.2) to reduce the size of the returned file. Data filtering is an integral part of the Savant concept detailed in section 2.3.3. If we filter useless data from desired data before we put it on a transmission wire, we can cut our bandwidth usage greatly.

**Massive distributed caching** The sheer size of the target implementation of the Auto-ID system would overwhelm any one central data server, and so the second step we take to deal with high bandwidth is making the system highly distributed and capable of caching data locally. If data requests are divided amongst N servers, we can divide the bandwidth required by any given server by N. Also, if we only have to ask for a given piece of information once, instead of multiple times, we again cut down heavily on the bandwidth required to implement this system. As we discuss in section 2.2, DNS is already a distributed system that deals with caching on its own. We cache PML data within most applications of this system, as detailed in section 3.
Overall general conclusions We need a distributed, integrated, low bandwidth system to translate the unique identifier in each ePC tag into information about that product. This system must be robust and it must be capable of handling the serving of information for trillions of tags. We have the tools of data filtering, distributed server engineering and intelligent caching of information to apply to this problem. While security is an issue, existing security systems will be easily adapted in the future to this system once it is shown that other technical hurdles have been overcome.

2.2 Overview of Auto-ID Architecture

The costs of implementing [production-level DNS] dictate that it be generally useful, and not restricted to a single application.

We should be able to use names to retrieve host addresses, mailbox data, and other as yet undetermined information.

RFC 1034, DOMAIN NAMES - CONCEPTS AND FACILITIES[5]

Fortunately, there are systems in existence now that perform functions that are easily usable for our purposes. DNS, the Domain Name System, is a highly distributed system used for translating between 32-bit IP addresses and arbitrary length hostnames. The eXtensible Markup Language, or XML, is a robust and easily manipulable language that we can easily use to format data about tagged items[7]. Several XML query languages are also in existance, and while they are not evolved to the point where industry feels comfortable using them, we used one named XQL\(^1\) [6] to facilitate the data filtering required above.

2.2.1 ePC overview

The ePC [1] is a format for a unique identifier, or UID, for every RFID tag used in the Auto-ID system. Given the inexpensive and passive nature of the tags most

---

\(^1\)XQL stands for XML Query Language. There are others - there is one intuitively named XML-QL, and another named Quilt. XQL was chosen because there was a Perl module on CPAN that supported XQL queries on XML files.
Figure 2-1: An overview of where various subsystems fit in the Auto-ID system
likely to be used by the Auto-ID center, the ePC has been designed to work with tags that store on the order of 128 bits. The basic ePC requires a minimum of 96 bits, separated into 4 required and 1 optional field:

- ** Bits 0-7:** (8 bits) The **Header** section is required to differentiate the ePC format from other RFID UID formats.

- ** Bits 8-35:** (28 bits) Each **Manufacturer** is assigned a 28 bit code to uniquely identify their products. This 28 bit block allows for dataspace partitioning in a convenient way for ONS (section 2.2.2).

- ** Bits 36-55:** (20 bits) Every manufacturer assigns every **Product** they produce a 20-bit identifier.

- ** Bits 56-95:** (40 bits) Every instance of a product has a **Serial Number**, to uniquely identify that instance of that product.

- ** Bits 96+:** (0+ bits, optional) The **Product Data** section allows the manufacturer to encode object-specific information on the tag itself. In the ePC standard, this can be omitted.

Even with a fixed header, with $2^{88}$ unique identifiers available to the products served by the Auto-ID system, it is extraordinarily unlikely that Auto-ID will run out of namespace within the lifetime of the project.

### 2.2.2 ONS overview

The purpose of ONS is to efficiently convert an ePC into a location on the Internet that serves PML files relevant to that ePC. Given that there are $2^{88}$ possible ePC codes, the ONS system must be scalable, and given the intended global use of the Auto-ID system, the service must be fast.

Rather than write a fast, scalable, multitiered name resolution service from scratch for the Auto-ID system, we chose to use an existing system - DNS, the Domain Name Service. DNS maps hostnames (e.g. omaha.mit.edu) to IP addresses (e.g. 18.80.0.106)
and vice versa. The Auto-ID ONS system treats the ePC code as a hostname – currently, Auto-ID uses the realm “.objid.net” – and is therefore able to use the existing DNS system on the Internet, mapping a UID to an IP address instead of mapping a hostname to an IP address. Advantages to this system are many. The specific IP address serving a given ePC can change over time, and it becomes easy for a manufacturer to maintain ONS servers and serve PML files for its products.

2.2.3 PML overview

PML, which stands for Physical Markup Language, is an extension of XML, the eXtensible Markup Language. PML is currently in an extreme prototypical stage, as the Auto-ID center’s PML group is still trying to decide what sorts of data a PML file should hold about an object. Currently, the PML file is seen as holding several types of data.

Global Data The vast majority of tagged items will have certain attributes that will never change, and are the same for all individual copies of that type of item. In addition, some of these attributes will be common to all tagged items, such as weight, item type, and possibly MSRP\(^2\).

Static Data Many tagged items will individually have data pertaining to them that does not apply to other items of the same type. Expiration date, lot number, and date of manufacture are examples of this type of static data. This information will not change, but it is not covered by the above “global” data.

Dynamic Data Some tagged items will have data that changes from day to day – or second to second. In fact, most items will be able to make use of a type of dynamic data, such as location, in case a manufacturer of that item wishes to recall certain or all instances of the item. Dynamic data would also help for item location tracking in warehouses or shipping.

\(^2\)Manufacturer's Suggested Retail Price
Proprietary Data  The Auto-ID PML group will not be able to foresee all data a manufacturer may wish to make available about a tagged item. XML can reference other files within an XML file in a rather seamless manner, and this feature will be retained within PML. Proprietary PML files will be able to be referenced from within the “standard” Auto-ID template when it is published.

2.2.4 Savant overview

A “Savant” is the Auto-ID term for the lynchpin automated process that performs all of the assorted cross checks and informational lookups connected with converting an ePC into data from a PML file in a fast, bandwidth-efficient manner. There are technically two classes of savant, classified by their passive or active natures. A passive savant is called an “idiot-savant” and performs most of the same functions as a savant, but is not an active process.

The Auto-ID definition of a savant is a process connected to at least one Auto-ID capable network which sends synthesized data to the “outside world” without being polled for that data. [10] It usually, but not always, has more than one “network”, and acts as a gateway from its local network to the “outside world”. The savant is an inference engine - it synthesizes new data from data it can access, on its local network or on an outside network. It then sends this synthesized data to somewhere else.

Making the default inference engine a network gateway allows us as system designers to filter data - allowing only what data we wish to be accessible to the outside world and pass the savant do so. A gateway is also a natural place for security and access control. Making the default “gateway daemon” an active system promotes the massive distribution of data synthesis. An active system can process commonly-requested information in parallel and preemptively. Naming this engine a “savant” allows the Auto-ID center to discard previous terminology and create our own, such that when the Auto-ID system goes “live”, we do not inadvertently confuse our customers and users by referring to a confusing term such as “integral gateway daemon”.

21
In addition, note that the “outside world” specified above does not need to be contacted through a network. The MP3 player described in Chapter 4 is a savant - it actively polls a tag reader and when a tag is detected, outputs data synthesized from the ePC on that tag.

2.2.5 XQL queries in PML requests

With all of the data that will likely be included in the “standard” Auto-ID PML specification, it behooves us as system designers to include any way possible of decreasing the bandwidth required to support this system. One way the Auto-ID system will do this is to prefilter PML responses from PML servers with an XQL query.

XML uses tags much like HTML. However, in XML the content of the tags is up to the designer of the document. XML uses a concept known as a Document Type Definition, or DTD, to ensure “proper” formatting of a file, and thus an XML (or PML) file can be laid out any way its designers wish to write the DTD. This gives the XML file a tree-like node structure, with each set of tags encapsulating another node of the tree.

XQL is designed to take advantage of this tree structure. An XQL query will return one or more of these nodes. With a well formed DTD and an idea of the information one actually needs, an XQL query can be used to cut away a great deal of unneeded information in a large PML file, returning to the querying savant only what that savant needs for its data synthesis purposes.

2.2.6 RFID Tags

Motorola Corporation has been supplying the Auto-ID center with passive low-price RFID tags capable of holding 40 and 96 bits of information, as well as RFID tag readers for these tags. While the tag price currently is on the order of $0.25 per tag,
Motorola is working to reduce this price, and is shipping the Auto-ID center LonTalk-capable RFID tag readers to interface with our local network choice more readily. The Auto-ID center has been working with many different types of Motorola readers, as well as a few non-Motorola systems, and as such any details on the RFID tag readers will be covered in the actual descriptions of the prototype savants in Chapter 4.

2.3 Current Local Network Paradigm

The current local networking paradigm revolves around a “savant”, our form of gateway, and the network-capable items on the “back side network” of this savant. There are 7 distinct types of device in our system: virtual databases, physical databases, mechanisms, idiot-savants, savants, fabricators, and workshops.\(^3\)

2.3.1 Query/Response System

In our system, we have a number of devices which accept XQL read and write queries (“queries” and “requests” in our system) and respond to said queries. On the outside these systems can all be treated as simple databases; however, in implementation they are significantly different and as such deserve subcategorization.

- Virtual database

  Our first type of device is a simple XQL-compliant database. We call this type of device a “virtual database” to differentiate it from our “physical database” device category. This device has no program state and only responds to XQL queries and requests.

- Physical database

\(^{3}\text{For the Auto-ID system, we decided to discard previous terminology and create our own, neutral, set of keywords. In this way, we hope to avoid confusing our customers and beneficiaries when the system goes "live".}\)
Device Classification for Auto-ID connectivity framework

Figure 2-2: The 7 types of device in our system
Our next type of device is a "physical database" - a mechanical/electrical system of some sort that responds to queries and requests with information. An example of a physical database would be a temperature sensor that responded to a database lookup of its temperature. This type of device treats the physical environment that it is reading through its sensory mechanisms as its "data space". Our ideal physical database has no "program space" - we simply issue queries to the physical database device to find information in the physical environment that the device uses as its database.

- Idiot-Savant

One of the desired functions of our devices is the ability to produce an inference engine: namely, a device on the network that can have a set of data input and from that set infer another piece of data, which can then be queried, all using our XQL query/request framework. We have two devices which can fulfill this role: the savant and the idiot-savant. The savant is described below and is a superset of the idiot-savant.

The idiot-savant is a passive device that can make limited inferences on data input via XQL queries/requests. The idiot-savant can have an additional network connected to it - the synthesis of the idiot-savant and this additional network is referred to as a fabricator (a passive version of the workshop, detailed below). On the back-side (internal) network of a fabricator, there can exist any number of devices to assist the idiot-savant in its task of generating inferences from XQL queries/requests sent to it. Note that the idiot-savant is not an "active" device - it infers and/or synthesizes data when it is queried, not on its own.

2.3.2 Mechanism

One of the primary desired functions of this project is to provide connectivity to mechanical devices. A "mechanism" is a device in our system that only accepts requests - a write-only device. The desired niche of the "mechanism" device is to serve as a pattern for interfaces to mechanical devices - such that one can issue commands
Figure 2-3: The path of a full ONS/PML query

to mechanical devices hooked up to our network. For example, a command could be issued to turn on a fan or other mechanism, but a query as to whether the fan was on or off would be ignored (and most likely would not even be supported). That query would also be meaningless if the fan was connected to the network as a simple mechanism, as the fan-mechanism could neither respond as to its own state nor would it have any state in which to record whether it was “off” or “on”. Any device that would respond in that situation is effectively a fabricator (see idiot-savant above in Section 2.3.1) with a mechanism (and perhaps a static database of some sort) on its internal network.

### 2.3.3 Savant and Workshops

The savant is the central “active” component of our classification system. A savant sends synthesized data to the outside world without being polled for that data. It usually has more than one network, and as such acts as a gateway from its local network to the outside world. Aside from the active nature of the savant, it behaves quite similarly to the idiot-savant described in Section 2.3.1. A workshop is a savant with an internal network, the active analogue to a fabricator.
2.4 Path of a Full Query

Savant retrieves ePC identifier  To begin the query, the savant must acquire the unique identifier housed within the ePC “tag”. Usually this is done by means of a tag reader within the savant’s workshop - the reader itself would be a “physical database” as explained above.

Savant converts ePC ID to PML server using ONS  ONS is an application of DNS, and as such retains DNS’s local caching ability. Therefore, the first step we take in finding the ePC’s PML server is to check the ONS cache. If it is not there, we then send a request to lookup the PML server’s hostname as detailed in Section 2.2.

Savant queries PML server for PML file  Once the savant has the location of the PML server, it can query the server for the PML content it requires. Most savants, however, will likely maintain a PML cache (detailed in Chapter 3). Therefore the savant would check this first and, if the data in the cache pertained to the relevant ePC and was not expired, would fail to make the query.

If the savant still needed to query the PML server, it can use an XQL filter to reduce the size of the file sent back so as to reduce the workload of the server.

PML server returns the requested information to the savant  The PML server retrieves the PML file from wherever it is stored and filters it with any XQL filter enclosed in the query before returning the data to the savant.
Chapter 3

Savant Specification

The Auto-ID system is an integration of a large number of disparate subsystems, which were described briefly in Chapter 2. However, the main work of this thesis is the savant, the local network controller at the heart of the myriad distributed systems that the Auto-ID Center wishes to promote. As such, this chapter discusses the savant in detail. Below, I list the most likely tasks that a savant will need to perform. [9] I also provide a list of subcomponents that an “ideal” savant and idiot-savant should incorporate, in preparation for the prototype savants presented in Chapter 4.

3.1 General Classes of Data Queries

A savant is in essence a networked database with a number of active and inferential components. Some savants are gateways, some savants will manage a large number of physical devices (tag readers), and some savants will simply compile and publish data, but all savants provide data, and as such can be abstracted as databases. It is useful from an architectural standpoint to categorize the general types of queries we expect these databases to see.
3.1.1 Virtual Database Queries

The savant supports standard databases in the form of a “virtual database” device. This is perhaps the most mundane type of query we can expect, however it will likely make up the bulk of operations (possibly vying with physical-space database queries, section 3.1.2) and is a necessary component of the inferences that will be needed to respond to the other types of queries below. Note that a “database” does not necessarily need to be a simple application on a machine somewhere - sensors provide data, and thus can be considered databases. In addition, databases may update their own data dynamically (sensors again are a useful example of this) and so a database need not be static.

3.1.2 Physical-space Database Queries - Physical Databases

One of the primary components of any Auto-ID system will be the Auto-ID tag, a wireless system on a single chip that contains and can transmit a unique identification code\(^1\) to an Auto-ID tag reader. With strategically placed tag readers and tags on all inventory items, a warehouse can be transformed into a physical-space database where the data stored are the actual inventory items themselves. Objects can be moved in real time. Data is stored by the physical location of items and is retrieved by querying a tag reader. The tag number can then be looked up via ONS and PML servers to provide more information on the object.

3.1.3 Dynamic Database Requests - Mechanisms, Savants and Idiot-Savants

Our system of devices generates and receives requests - a “write” or “execute” command - in addition to simple stores. For example, a mechanism could be controlled remotely (or by the controlling savant in the workshop the mechanism is installed in)

\(^1\)At the time of this proposal the tag specification is changing - the length of the ePC may remain at 96 bits or it may grow.
with an XQL request. A savant could have several savants under its control (in its workshop), working as a distributed system. An active savant controlling a workshop could make a request of a database located on its external network to retrieve data pertaining to a tagged inventory item in a physical database in the workshop managed by the savant.

With requests of this nature come the need for security and authentication. Our system will need to make sure that anyone making a request for information has the appropriate access to do so. In fact, the concept of the Savant provides three important ways to streamline handling requests of this type.

- **Distribution of Data Synthesis** - Myriad Savants, each with its own workshop, can each perform data synthesis such that the overall task of fulfilling a complex dynamic database request may be simplified.

- **Data filtering** - All of the data and devices on every local network does not need to visit the outside world or be accessed by users, whether benign or malicious. The "gateway" nature of the savant allows us, as system designers, to limit access to the local networks.

- **Security** - A Savant is a free gateway and firewall, and it is trivial to prevent access to undesireable elements. If we wish to provide access to some items within the local network to some users while denying it to others, the Savant is more than capable of dealing with the required access control.

### 3.2 Characteristics of an ideal idiot-savant

In order to keep the savant paradigm robust, an idiot-savant has a necessarily simple minimum requirement - that of a data synthesis engine. When queried, an idiot-savant must return synthesized data - data that did not originate in the idiot-savant, but was synthesized from another source. A degenerate idiot-savant, therefore, could
be a simple gateway that queries a device on one network for data in response to a query from another network.

However, the purpose of this section is to show an “ideal” implementation for the uses in which the Auto-ID project foresees using idiot-savants. A fast, efficient, distributed architecture is needed for the Auto-ID project, and thus our “ideal” idiot-savant will have several integral caches and databases to facilitate the smoother functioning of the system as a whole. In addition, some of these subsystems will not be needed for every actual savant or idiot-savant, and so the subsystems should be modular - a savant designer should be able to insert or remove subsystems until they achieve a desired configuration.

**Local network of RFID tag readers** The Auto-ID system revolves around the ePC and the RFID tags that ePCs are typically read from. An ideal idiot-savant should have the capability to converse with a network of “standard” Auto-ID RFID tag readers.

**ONS Cache** An idiot-savant will most likely need to resolve ePCs into PML servers. DNS supports a local cache in every intelligent implementation of a DNS client (the data itself is inherently cacheable, even if certain implementations\(^2\) do not cache the DNS data). As ONS is built directly around DNS, one needs only incorporate a simple intelligent DNS client (Unix named, for example) to accomplish ONS caching.

**Local PML Cache / Local DBMS** Savants and idiot-savants are both inference engines; they synthesize new data from old data. To store enough old data to synthesize useful new data, the ideal idiot-savant should have an integral local database. In addition, the idiot-savant may wish to interface this data with other services, like a PML server (below), and an idiot-savant may need to cache PML data to lower bandwidth usage of the system as a whole. The ideal idiot-savant would therefore benefit from containing an integral low-cost low-maintenance DBMS - MySQL, for

\(^2\)e.g. Microsoft Windows.
example - to keep track of PML files downloaded from PML servers relevant to ePCs read by the aforementioned network of readers, as well as data locally gathered by the idiot-savant, as well as any other data relevant to the idiot-savant. If the idiot-savant is acting as a savant (or is serving PML data), the DBMS can also store information used by the PML server, below.

3.3 Characteristics of an ideal savant

In addition to the data synthesis requirement of the idiot-savant, the savant must also send this data to the outside world somehow. This may be as simple as actively sending music over a speaker, or it may be a complex schedule of sending data to several master servers on the Internet. While (as in the case of the idiot-savant) there is a degenerate and trivial solution to making a “savant”, this section will detail suggested components of an ideal savant for the applications a savant is likely to need to perform in the Auto-ID system above and beyond those that an idiot-savant can perform.

XQL-enabled PML Server Some savants will be able to update the PML files of certain ePCs. In addition, as PML is the standard informational exchange format of the Auto-ID project, savants that do not necessarily have access to manipulate the actual “authoritative” PML file for a given ePC will still have cause to publish PML globally about these ePCs - a certain set of ePCs are in inventory in a given store, or have been registered as “destroyed” in a certain facility, and so on. For both of these applications, it would be useful and expedient for an “ideal” savant to have a built-in PML server.

In addition, due to the overriding desire to limit bandwidth, any savant that boasts a PML server should have an optional XQL filter implemented, such that anyone requesting PML data from the server can download only what they need (as designated by an XQL filter sent along with the PML query). However the PML server for that savant is implemented, the XQL server should be implemented “on
top” of the PML generation/retrieval routines, such that the PML file can be run through the XQL filter before being sent over the Internet (or whatever transmission medium is being used.)
Chapter 4

Savant Implementations

While in the last chapter I discuss the “ideal” savant and idiot-savant, in this chapter I present two prototype systems, one savant and one idiot-savant. First I detail some of the component systems of these prototypes, and then I explain why each prototype was constructed in the way that it was. Finally, I explain what was learned from the construction of each prototype and how each prototype could have been improved.

4.1 Overview of Component Systems

The savant implementations described in this chapter are not written completely from scratch. Most of the savant is made up of pre-existing components, and the rest of the savant is simply “glue” code to make the disparate parts work together in a reasonably seamless fashion. In this section I describe the main pre-existing systems that I used in the prototypes in this chapter and hope to see in use on future production-quality savants put out by the Auto-ID center.

4.1.1 MySQL

Capabilities and brief overview of MySQL MySQL is a relational database management system, or RDBMS, distributed under a very liberal Open Source copyright. Due to its open source nature, it was a natural choice for a massive system
that could not support a more expensive DBMS like Oracle due to the sheer number of databases that would need to be maintained. [11]

MySQL is a multithreaded system, making synchronous access by multiple parties simple and painless. It has always supported a subset of SQL, however, this subset includes most useful parts of SQL, omitting stored procedures and transactions. The current beta release of MySQL supports transactions with minor bugs, and so by the time that the Auto-ID system is entering a distribution phase, we can most likely count on MySQL transaction support. In addition, MySQL supports a few small extensions of ANSI SQL - the most useful of those used in the prototype systems being the AUTO_INCREMENT data attribute. With these features, MySQL is a reasonably powerful and inexpensive choice for a DBMS for prototype savants.

**Shortcomings of MySQL** Given the preponderance of PML and XQL in the Auto-ID system, it is unfortunate that MySQL does not currently have the capacity to output information in an XML format or to accept an XQL query directly. However, the field of XML query languages is a new one. The current market leading database system, Oracle, does not accept XQL queries, simply because the industry has not come to a consensus on which XML query language to support. As such, the Auto-ID project needed to decide on an XML query language, and decided on XQL due to the Perl library support for it detailed in section 4.1.3.

Another potential problem with MySQL is its status as a relational database management system. Most commercial database management systems today are taking an object-oriented approach, and the SQL3 standard is slowly migrating to the concept of an object oriented DBMS. If the Auto-ID project is successful enough, it may revitalize the market for a simple, no-frills RDBMS - however, it may be just as likely that MySQL will be swapped out for a cheap, no-frills OODBMS. This does not, however, invalidate the use of MySQL as the DBMS for prototype Auto-ID savants.
4.1.2 LonWorks

The Auto-ID project needs a protocol and framework for its local networking solution. Several (approximately 250) manufacturers have joined together in the LonMark coalition and have produced products capable of communicating over a LonWorks network with the LonTalk protocol. This section details important aspects of LonWorks, such as the protocol, what you can do with it, how much it costs, and what the Auto-ID center would be using it for.

Capabilities and brief overview of LonWorks  LonWorks is to be used as a local networking layer in the Auto-ID system. [8] It will network tag readers, appliances, and computers running savants together. It is my intention to restrict access to the Internet at large to savants and idiot-savants, so any appliance, tag reader, et al that needs to communicate with the world at large will have to send its data over a LonWorks channel first.

The LonWorks protocol is outlined in the ANSI/EIA 709.1 Control Networking Standard, and is explained in several Echelon documents. [2]. (Up until recently, however, the protocol was proprietary and was only available embedded in the Neuron Chip, detailed below.)

Addressing  There are four ways to address a LonWorks device.

- **Physical Address** The first is with its physical address, a unique 48-bit identifier called the Neuron ID. This is set when the device is manufactured, like a MAC address for Ethernet, and does not change.

- **Device Address** A LonWorks device is assigned a device address when installed in a network. Device addresses are used instead of physical addresses because they support the more efficient routing of messages along the network. Device addresses consist of: **domain ID, subnet ID** and **node ID**. The domain is a group of devices which may interoperate - multiple domains can exist on the
same physical network independent of each other. The subnet is a collection of up to 127 devices on a single channel - see “Broadcast Address” below. The node ID identifies a single device within a subnet.

- **Group Address** A LonWorks group is a logical grouping of devices within a domain. Unlike a subnet, however, they do not need to be on the same wire. There can be any number of devices in a LonWorks group if unacknowledged messaging (analogous to UDP, described below) is used; however, if using acknowledged messaging (analogous to TCP), there are a maximum of 63 devices per group.

- **Broadcast Address** Much like IP, LonWorks supports the concept of a broadcast address. This transmits to all devices within a subnet, or all devices within a domain, depending on the specific broadcast address.

**Messaging** The LonWorks protocol supports several messaging modes.

- **Acknowledged Messaging** This provides for end-to-end acknowledgement, handshaking, and theoretical lossless transmission of information. Messages are sent to devices or groups of up to 64 devices.

- **Unacknowledged Messaging** This provides for messaging with very low overhead. A message is simply sent and no acknowledgement is given or expected.

- **Repeated Messaging** This provides a bit of backup to unacknowledged messaging. A message is sent a number of times, however, again, no acknowledgement is given or expected.

- **Authenticated Service** All three types of messaging support authenticated service, to see if a sender of a message is authorized to send that message.
Components of a LonWorks network  A LonWorks network consists of a set of devices networked together on a transport media. Devices can be subdivided into subnets and groups, as detailed above, and repeaters can be set up between networks.

Table 4.1: LonWorks Theoretical Maximums

<table>
<thead>
<tr>
<th>Networking</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Devices per Subnet</td>
<td>127</td>
</tr>
<tr>
<td>Subnets per Domain</td>
<td>255</td>
</tr>
<tr>
<td>Devices per Domain</td>
<td>32385</td>
</tr>
<tr>
<td>Domains per Network</td>
<td>$2^{48}$</td>
</tr>
<tr>
<td>Maximum Devices per System</td>
<td>32385 * $2^{48}$</td>
</tr>
<tr>
<td>Members in Acknowledged Group</td>
<td>63</td>
</tr>
<tr>
<td>Groups per Domain</td>
<td>255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet Size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes per Network Variable</td>
<td>31</td>
</tr>
<tr>
<td>Bytes per Application Message</td>
<td>228</td>
</tr>
<tr>
<td>Bytes per Data File</td>
<td>$2^{32}$</td>
</tr>
</tbody>
</table>

Table 4.2: LonWorks Transport Media Bandwidth

<table>
<thead>
<tr>
<th>Channel Type</th>
<th>Medium</th>
<th>Bit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP/FT-10</td>
<td>Twisted pair</td>
<td>78kbps</td>
</tr>
<tr>
<td>TP/XF-1250</td>
<td>Twisted pair</td>
<td>1.25MBps</td>
</tr>
<tr>
<td>PL-20</td>
<td>Power line</td>
<td>5.4kbps</td>
</tr>
<tr>
<td>IP-10</td>
<td>LonWorks over IP</td>
<td>Per IP medium bandwidth</td>
</tr>
</tbody>
</table>

Neuron Chip  The Neuron Chip is the heart of most commercially available LonWorks devices. In essence, it is an 8-bit microprocessor with the LonWorks protocol hardcoded onto it, programmable in Neuron C (a superset of ANSI C). The additional functions of Neuron C allow it to serve as an event-based language, which will make savants and idiot-savants easy to code directly into embedded systems in the future. The Neuron Chip is relatively cheap, with some implementations costing less than $3 USD\(^1\). The Neuron chip implements the bottom 6 layers of the OSI 7 layer paradigm

\(^1\)An Echelon sales representative quoted the price for the Neuron chip, in quantity, at $3 to $7 per chip.
for LonWorks in firmware, leaving the application layer to the LonWorks developer.

4.1.3 Perl Modules

All of the prototype savant code detailed in this thesis is written in Perl 5. Several public-domain Perl modules were used to facilitate easy prototyping. Various XML tools were invaluable in parsing XML and resolving XQL queries, and the Enhanced CD Player savant controlled a Motorola tag reader through a serial port. The Intelligent Shelves idiot-savant was intended to use the same serial port code.

Perl is a very versatile language for the purpose that I used it for in this thesis - prototyping systems. While it is not in my opinion a viable production language, the two savants detailed in this thesis are proof-of-concept designs, and as such Perl is a reasonable language for them to be written in. Perl also lends itself very well to CGI scripts for fast World Wide Web prototyping, and Perl’s facility to this made it easy to implement savants and PML servers that used HTTP to communicate over the Internet.

**libxml** The Auto-ID project uses PML, a markup language which is essentially XML verified by a DTD still under construction, to hold, organize, and transfer data relevant to ePCs in the Auto-ID system. Instead of spending a great deal of time writing XML and PML parsing, verification and handling routines in Perl, I simply used several public domain² Perl modules that manipulated XML and XQL. libxml-perl-0.05 served for using existing HTML-parse routines on XML. While this was somewhat useful in its own right, it mainly served as a prerequisite for libxml-ennos-1.00[3]. libxml-ennos-1.00, written by Enno Derksen, served for making XQL queries on XML documents. The XQL routines in particular were useful for constructing an XQL filter on the PML server aspect of the Intelligent Shelves prototype idiot-savant.

**Device-SerialPort-0.07** While the Auto-ID project is evaluating the use of Lon-Works as a local network protocol, the earlier prototype savants were constructed

---
²A huge number of Perl modules are available on CPAN (www.cpan.org).
using simple RS232 serial communications as the “local network”. (This obviously had a severe fanout constraint, and as such was just used for proof-of-concept savant software.) To interface with the serial port on the test platform (Linux 2.2.12 on an i686-based machine) I used the CPAN Device-SerialPort-0.07 Perl module. The use of this module can be seen in the Enhanced CD Player savant, as the tag reader in that prototype communicates with the rest of the savant through the serial port.

4.2 Enhanced CD Player

The first prototype savant constructed for the Auto-ID project was the Enhanced CD Player. In essence, the Enhanced CD Player was a theoretical attempt to redefine the way in which music would be purchased, using the MP3 music file format and an ePC on an RFID tag in every CD case. A user “scans” a CD and the savant plays the MP3s designated in the database to be on that CD. The Enhanced CD Player was intended to be expanded. A music publisher would produce a PML file for each ePC-tagged CD they sold (per the Auto-ID paradigm). Each PML file would have pointers to MP3 versions of the songs on that CD. In addition, the music publisher could add songs to a CD after publishing. The Enhanced CD Player savant would proactively compare these pointers in the online PML files to the MP3s in its own database, and download any new songs. In essence, a CD becomes a both a backup
of musical data and a subscription to a musical publication.

In both this example and the next, the UIDs on the RFID tags were not 96 bit ePCs, but 40-bit serial numbers. However, the Enhanced CD Player is a proof-of-concept prototype and it is unlikely this will be a problem. In addition, the MP3 comparison and active downloading was not implemented in this proof-of-concept savant. However, the aforementioned background is relevant to the way the savant was constructed.

The Enhanced CD Player consists of a Motorola RFID tag reader reading Motorola Bistatix passive capacitive 40-bit RFID tags, connected to the serial port of the machine running the actual savant. The savant machine also runs a MySQL database containing data pertinent to both the savant and a PML server run by the savant. The savant continually polls the tag reader for the presence of any tags. If a tag is detected, the tag reader queries its local database for information on the ePC on that tag. The database keeps a “playlist” for each CD with pointers to all of the MP3s containing the music for that CD. When a valid CD tag (i.e. one in the database) is scanned, the savant plays the MP3 files on that playlist in a random order using the utility “mpg123”.

If an invalid CD tag is scanned, the savant ignores it. In an ideal implementation of the Enhanced CD Player, the savant would perform an ONS/PML lookup of the CD tag and begin downloading the MP3s that the CD’s PML file listed. However, this feature is not implemented. The Enhanced CD Player’s MySQL DBMS does not include a PML cache for this reason - a local PML cache is not necessary because this savant is not downloading information about any ePCs.

### 4.2.1 MySQL Database Table Setup

Some of the parts of the CD database structure were never used - anything relating to PML servers, for example, was never implemented. However, the database structure here and in the Intelligent Shelves prototype should give a good idea of how to implement a standard savant “local database”, storing ePCs, data, and PML information.
song.song.fingerprint was intended to be used to verify uniqueness of mp3 files through the use of an MD5 hash or similar fingerprinting mechanism. It was never used in this prototype.
mysql> show tables;
+------------------
| Tables in cd    |
+------------------
| cd              |
| playlist        |
| playsong        |
| song            |
+------------------

mysql> show columns from cd;
+-------------------------------+---------+---------------+-------------------------|
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdid</td>
<td>int(11)</td>
<td>I</td>
<td>PRI</td>
<td>0</td>
<td>auto_increment</td>
</tr>
<tr>
<td>cd_tag</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>cd_name</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>cd_website</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
+-------------------------------+---------+------|-------|---------|----------|

mysql> show columns from playlist;
+-------------------------------+---------+---------------+-------------------------|
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>playlist_id</td>
<td>int(11)</td>
<td>I</td>
<td>PRI</td>
<td>0</td>
<td>auto_increment</td>
</tr>
<tr>
<td>playlist_name</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
+-------------------------------+---------+------|-------|---------|----------|

mysql> show columns from playsong;
+----------------+--------------+---------+---------------+-------------------------|
| Field          | Type         | Null    | Key   | Default | Extra    |
|----------------+--------------+---------|-------|---------|----------|
| playsong_id    | int(11)      | I       | PRI   | 0       | auto_increment|
| song_ref       | int(11)      | YES     |       | NULL    |           |
| playlist_ref   | int(11)      | YES     |       | NULL    |           |
+----------------+--------------+---------+-------|---------|----------|

mysql> show columns from song;
+----------------+---------------------+---------+---------------+-------------------------|
| Field          | Type                | Null    | Key   | Default | Extra    |
|----------------+---------------------+---------|-------|---------|----------|
| song_id        | int(11)             | I       | PRI   | 0       | auto_increment|
| song_name      | varchar(255)        | YES     |       | NULL    |           |
| song_filename  | varchar(255)        | YES     |       | NULL    |           |
| cd_ref         | int(11)             | YES     |       | NULL    |           |
| song_fingerprint | varchar(255) | YES |       | NULL    |           |
+----------------+---------------------+---------+-------|---------|----------|
4.2.2 cd-input.pl

This piece of code (Appendix A.1 was used to load up the test database (see Section A.7). I used a pretty standardized approach to dealing with the MySQL database throughout the entire project. At times, this led to me performing computation within Perl that could have been easily done within SQL; however, as this was a proof-of-concept design I did not think much of it at the time.

4.2.3 scan-CD.pl

scan-CD.pl (listed in Appendix A.4) is the core of the Enhanced CD Player savant. It polls the serial port every half second for a UID from the Motorola tag reader, and if it receives one, it looks it up in the MySQL database detailed above in Section 4.2.1. If the UID denotes a valid CD playlist, scan-CD.pl retrieves the playlist and feeds the file pointers to an mpg123 process, which proceeds to play all the MP3s in a random order. scan-CD.pl then resumes scanning the serial port for UIDs.

Main program loop

- poll serial port
- if tag reader has read UID
  - Check for valid playlist matching UID in MySQL database
  - if valid playlist
    run MP3 player on playlist
- sleep for 500 milliseconds
- loop

Currently scan-CD.pl queues all UIDs sent through the serial port while mpg123 is playing a series of MP3s. A trivial modification would be to purge the serial port buffer, so as to not get accidental repeat scans of CDs.
4.2.4 get-playlist.pl

This was an intermediate piece of code (listed in Appendix A.2), which did not poll the tag reader for input but relied on the user to request a playlist. It could certainly still be used as a command-line interface to the Enhanced CD-player savant, but the user would have to know the UID stored on the tag of the CD he wished to play.

4.2.5 punt-playlist.pl

This piece of code (listed in Appendix A.3) was another administrative tool. It allowed the user to purge a playlist (or CD) from the Enhanced CD Player’s MySQL local database. This code is included like most of the other administrative/command line tools for completeness’ sake.

4.2.6 playlist-CD.pl

playlist-CD.pl (listed in Appendix A.5) is a command-line interface that was a pre-cursor to scan-CD.pl. If the user knows the UID (or ePC) of the CD he wishes to listen to, he can use playlist-CD to retrieve the playlist of that CD and listen to it without using scan-CD.pl and scanning the CD. Again, this code is included for completeness.

4.2.7 CGI script: http://omaha.mit.edu/cgi-bin/PML

This was a simple proof-of-concept program (listed in Appendix A.8) that allows you to download a “PML file” about any of the CDs in the CD database from the omaha.mit.edu webserver. The downloaded PML file contained the name of the CD (as contained in the Enhanced CD Player database) and the RFID tag UID queried. If the tag UID was invalid or if it did not correspond to a CD in the database, the PML server returned an empty string for the name of the CD.
4.2.8 Enhanced CD Conclusions

The Enhanced CD Player shows that it is plausible to construct a savant that actively polls and responds to changes in its local network of tag readers. The Enhanced CD Player implemented a local MySQL DBMS and used it to good effect, and controlled a degenerate local network of one RFID tag reader. It also provided a PML server for outside parties to examine some of the contents of the local CD playlist database.

The Enhanced CD Player could have been improved with a network of tag readers and a more comprehensive PML server. In addition, none of the “outside world” infrastructure for the Enhanced CD Player - PML servers publishing music data in MP3 format - was implemented.

4.3 Intelligent Shelves with LonWorks Readers

The Intelligent Shelves project was an attempt to make a set of shelves, equipped with RFID tag readers, accessible within the Auto-ID paradigm over the Web. A snapshot of the objects on a set of readers (the “shelves”) would be taken every so often and stored in a database with many other such snapshots, organized by timestamp.
These timestamps would be retained in the database at varying frequencies, with high accountability for the most recent snapshots and larger delays between snapshots in the older, stale data. Extremely old data would simply be purged. These snapshots could be accessed over the Web, and would be returned as a standardized PML file.

While for simplicity’s sake I will refer to the Intelligent Shelves as a savant in the rest of this section, it is important to note that it is really an idiot-savant, as it does not actively send any data to the outside world. If it, for example, played MP3s from the playlists in the previous section corresponding to a random tag at arbitrary times, it would be providing data to the outside world and would therefore be a savant.

The intent of the Intelligent Shelves was to be a testbed for a local network technology for the Auto-ID project. The technology to be tested was LonWorks, explained in Section 4.1.2. LonWorks was to be used to network several LonTalk-capable RFID passive tag readers. Unfortunately, the Auto-ID center was unable to acquire tag readers of this type in time for this thesis to detail them. As it is difficult to construct a local network with nonexistent peripherals, I have written a “black box” local network simulator that randomly generates data much like that which would be read off of a hypothetical local network of RFID readers.

The Intelligent Shelves savant is run by two main processes. The first is a cron job that randomizes the “BBox” black box local network every 3 minutes. The second is the timestamp.pl program, which runs every 5 minutes. timestamp.pl manages the addition of timestamp data to the local MySQL database, as well as purging old data from the database. While we are using the MySQL DBMS to store timestamps, we again do not need a PML cache, as as the savant is not downloading information about the ePCs returned by the simulated local network, but is instead synthesizing such data.

4.3.1 MySQL Database Table Setup

The field tag.server is the only field not used in this prototype’s database structure. In a system that downloaded PML files, an ePC would have its PML server resolved and cached in tag.server.
4.3.2 Virtual Tag Reader Network - BBox/randomize.pl

The virtual tag reader network, detailed in Appendix B.1, was a simple Perl script run by cron job to simulate a set of “shelves” (i.e. readers) with a high turnover of tags. randomize.pl reads the files locations and tags and randomly writes tag/location pairs to the file bbox.dat, which timestamp.pl accesses.
4.3.3 timestamp.pl

timestamp.pl (listed in Appendix B.2) was the core of the Intelligent Shelves savant. It performs the functions of entering timestamp data into the local database and filtering old data from the database. I save every timestamp for 24 hours, 1 out of every 2 timestamps for the previous 24 hours, and one out of every 4 timestamps for the previous 24 hours. Timestamps older than 72 hours are discarded. Since timestamp.pl is called from a cron job, it is a simple script that performs the above functions.

Main program sequence

- Read data from BBox/bbox.dat
- Create a new “timestamp” data structure in DB
- Insert new data into newly created timestamp structure
- if timestamp.age > 24 hours and timestamp.ID % 2 ≠ 0, purge timestamp
- if timestamp.age > 48 hours and timestamp.ID % 4 ≠ 0, purge timestamp
- if timestamp.age > 72 hours, purge timestamp

4.3.4 test-get.pl

test-get.pl (listed in Appendix B.3) was a simple PML “browser”, as Netscape wasn’t happy with showing XML. I used this completely for debugging purposes. If you need a quick hack to help debug implementations or improvements on this concept in Perl, this should help you out.

This “browser” takes in two arguments. The first argument is an XQL query, which the shelves’ PML server (below) dealt with server-side. The second is a timestamp to focus the search of the database. Both arguments were optional.
4.3.5 CGI Script: Savant

The “Savant” CGI script (listed in Appendix B.4) provides an interface over the Web to the Intelligent Shelves database of timestamps. It takes two arguments, a timestamp and an XQL filter. This script retrieves the data from the Intelligent Shelves database for the specified timestamp, or the most recent timestamp if no timestamp is specified, and formats it into a PML file. If an XQL filter is specified, the XQL filter is run over the PML file before the PML file is sent back to the querying host; otherwise, the newly-constructed PML file is simply sent in its entirety. This script demonstrates an XQL-capable PML-server front-end to the DBMS I intend to make standard in every idiot-savant and savant.

Since it is a script and not a daemon, this program simply executes a series of commands in order.

Main Program Sequence

- if time specified,
  retrieve data from database entry closest to but not after specified time
- else
  retrieve data from most recent database entry
- insert data into PML format, store in $PML
- if XQL filter supplied
  - filter $PML through XQL filter
  - return $PML
- else return $PML

4.3.6 Intelligent Shelves Conclusions

The Intelligent Shelves’ most impressive feature is the XQL-enabled PML server that publishes the timestamp data stored in the local MySQL database. XQL is not yet
an industry standard, but with applications such as the Auto-ID system it may grow to become the dominant XML query language. It is disappointing that the Intelligent Shelves did not actually implement a local network of RFID tag readers; however, it seems a reasonably uncomplicated task to produce a standard network of readers for a Shelves-like savant and tie them into the savant with code reuse from the Enhanced CD Player’s serial-port driver.

It is interesting to note that the PML server on the prototype idiot-savant is more advanced than the PML server on the prototype savant. This is an artifact of the order in which the projects were undertaken - I chose to include a more capable PML server in the Intelligent Shelves idiot-savant rather than simply forgo the XQL capability enhancements or retrofitting the Enhanced CD Player. It is a rather modular system in any case, and adding an XQL-filter capable PML server to the Enhanced CD player would be a matter of swapping out a few lines of code.

In both the Intelligent Shelves and the Enhanced CD Player, a local DBMS and a rudimentary PML server were easy to implement and became integral to the design of the savant. From this core, a number of applications could be performed by the addition of modular code fragments. A serial port driver and a callout to an mp3 player rounded out the enhanced CD player, while a simulated local network of readers and an XQL serverside filter rounded out the Intelligent Shelves. I believe the relative ease of assembly and modularity of code speaks well for the overall design of the savants presented here.
Chapter 5

Conclusions

The vision of the Auto-ID center is a world where all physical objects can communicate with each other any information one object might need on another. While we are unable to store all information any other object might need directly on an object, we can tag each object with a unique identifier called an ePC and store all relevant data to that object in a massively distributed system of databases that any other object can then access. This system of databases, as well as the means by which objects access them, is the Auto-ID system.

While the vast majority of the information about all of the world’s products will be stored in large PML databases maintained by item manufacturers, the Auto-ID system requires local control for scalability and performance. Every time an ePC needs to be translated into relevant data, a local system must make the translation from ePC to PML server and finally to usable data. There will be an uncountable number of local systems in the final implementation of the Auto-ID system, and these myriad local systems must be bandwidth-efficient or the entire system will die.

In addition, some local systems may synthesize data relevant to the rest of the Auto-ID project in some way. To control these data-generating local parts of the massive system and link them in with the global parts, a local gateway paradigm is needed. To manipulate and synthesize data relevant to the rest of the project, an inference engine paradigm is needed. A local gateway can provide security to the devices behind it, and can filter data so as to reduce bandwidth usage in general. An
embedded database management system can provide a powerful tool to manipulate data and synthesize new data.

All of these required capabilities - local control, local gateway, inference engine - are satisfied by the Savant.

In this thesis I detailed two prototype savants I implemented, the Enhanced CD Player and the Intelligent Shelves. The Enhanced CD Player was an active system that combined a database of music, utilizing the inference engine property above, with a local RFID tag reader, acting as a local gateway. The Intelligent Shelves combined local control over the logging of data with a simulated local network (therefore acting as a gateway) and an inference engine that could retrieve and manipulate the logged data. These three capabilities are complementary, and the manner in which they were coded into the two presented prototype savants was modular as well, which should aid in the construction of other savants for different applications. In short, the savant definition and the suggestions within this thesis work for the purposes of the Auto-ID system, and the prototypes within should serve as stepping stones for production-model savants when the Auto-ID system comes into widespread use.
Appendix A

Enhanced CD Code

A.1 Code: cd-input.pl

#!/usr/bin/perl -w use Mysql;

#get username password filename
$usage = "usage: cd-input.pl <user> <pass> <infile> [host] [db]\n";
$user = shift || die $usage;
$pass = shift || die $usage;
$inf = shift || die $usage;
$host = shift || 'localhost';
$dbn = shift || 'cd';

#connect
my $db = Mysql->connect($host, $dbn, $user, $pass);

#read file
open (IN, $inf) || die "File failed to open.\n";

my $cd_ref;
#my $cd_name, $cd_website, $cd_path, $cd_tag, $cd_ref;
#my $song_name, $song_filename, $song_fingerprint;

while (<IN>) {
    if ($_ =~ "\^CD") {
        #new CD entry
        chomp;
        (undef, $cd_name, $cd_tag, $cd_path, $cd_website) = split('\\', _);
    }
print "$cd_name $cd_tag $cd_path $cd_website\n";
$query = "insert into cd (cd_tag, cd_name, cd_website) 
    values ('$cd_tag', '$cd_name', '$cd_website')";
$output = $db->query($query);
print $output->as_string . "\n";
$query = "select cd_id from cd where (cd_tag = '$cd_tag')";
$output = $db->query($query);
$cd_ref = $output->fetchrow;
} else {
    #new song entry
    chomp;
    ($song_name, $song_filename) = split('\^', $_);
    $song_filename = $cd_path . $song_filename;
    print "$song_name $song_filename\n";
    my $query = "insert into song (song-name, song-filename, cd-ref) 
        values ('$song_name', '$song_filename', '$cd_ref')";
    $output = $db->query($query);
    print $output->as_string . "\n";
}

A.2 Code: get-playlist.pl

#!/usr/athena/bin/perl -w
use Mysql;

$usage = "usage: get-playlist.pl <user> <pass> <playlist-name>";
$user = shift || die $usage;
$pass = shift || die $usage;
$playlist_name = shift || die $usage;
$host = 'localhost';
$dbn = 'cd';

my $db = Mysql->connect($host, $dbn, $user, $pass);

$query = "select song.song_filename from playlist, playsong, song where 
    (song.song_id = playsong.song_ref) && 
    (playlist.playlist_id = playsong.playlist_ref) && 
    (playlist.playlist_name = '$playlist_name')";
$output = $db->query($query);

my @x;
while ($song_fn = $output->fetchrow) {
    print $song_fn. "\n";
    push @x, $song_fn;
}

$len = $#x;
$flag = 1;
while ($flag == 1) {
    $y = int(rand($#x));
    $blah = $x[$y];
    $z = 0;
    while ($z < $len) {
        if ($blah ne "used!!!") {
            system("/usr/bin/mpg123", $blah);
            $x[$y] = "used!!!";
            last;
        } else {
            $y++;
            if ($y == $len) {$y = 0;}
            $blah = $x[$y];
            $z++;
        }
    }
    if ($z == $len) {$flag--;}
}

print "Done!\n"

A.3 Code: punt-playlist.pl

#!/usr/athena/bin/perl -w
use Mysql;

$usage = "usage: punt-playlist.pl <user> <pass> <playlist-name>";
$user = shift || die $usage;
$pass = shift || die $usage;
$playlist_name = shift || die $usage;
$host = 'localhost';
$dbn = 'cd';
my $db = Mysql->connect($host, $dbn, $user, $pass);

$query = "select playlist_id from playlist where
(playlist.playlist_name = '$playlist_name')";

$output = $db->query($query);
$playlist_id = $output->fetchrow;
while ($blah = $output->fetchrow) {
    if ($blah !=$playlist_id) {
        print "\tThere's more than one playlist with this name.
\tI'm going to try to clean all of them.\n\n";
    }
}

if ($playlistid){
    $query = "delete from playsong where
(playsong.playlist_ref = $playlist_id)";
    $db->query($query);
} else {
    print "There didn't seem to be a playlist in there that was
named $playlist_name.\n";
}
$query = "delete from playlist where
(playlist.playlist_name = '$playlist_name')";
$db->query($query);

A.4 Code: scan-CD.pl

#!/usr/athena/bin/perl -w
use Mysql;
use Device::SerialPort 0.05;

$usage = "usage: scan-CD.pl <user> <pass>";
$user = shift || die $usage;
$pass = shift || die $usage;
$host = 'localhost';
$dbn = 'cd';

my $db = Mysql->connect($host, $dbn, $user, $pass);
#start serial input
my $file = "/dev/ttyS0";
my $pass;
my $fail;
my $in;
my $in2;
my @opts;
my $out;
my $loc;
my $e = 1;
my $tick;
my $tock;
local $ob = Device::SerialPort->new ($file) || die "Can’t open $file: $!";
$ob->baudrate(9600)  || die "fail setting baudrate";
$ob->parity("none")  || die "fail setting parity";
$ob->databits(8)     || die "fail setting databits";
$ob->stopbits(1)     || die "fail setting stopbits";
$ob->handshake("none") || die "fail setting handshake";
$ob->write_settings || die "no settings";

$ob->error_msg(1);  # use built-in error messages
$ob->user_msg(1);

local $query;

my $bop = 0;
while (1) {
    # timing loop
    # my $flag;

    # if we have something
    # if ($flag){
    #    get input
    my $gotit = "";
    until ("" ne $gotit) {
        $gotit = $ob->lookfor;  # poll until data ready
        die "Aborted without match\n" unless (defined $gotit);
        select undef, undef, undef, 0.5; # 10 polls/second, 960 bit maxlen
        # 240 bit suggested len (30 chars)

        print "$bop - \$gotit = $gotit\n";
        $bop++;
    }
    $cd_tag = $gotit;
}
$playlist_name = 'playlist_' . $bop;

$query = "insert into playlist (playlist_name) values ('$playlist_name');"; $db->query($query);
$query = "select playlist_id from playlist
   where (playlist_name = '$playlist_name');"; $output = $db->query($query);
$playlist_id = $output->fetchrow;
$query = "select cd_id from cd where (cd_tag = '$cd_tag');"; $output = $db->query($query);
$cd_ref = $output->fetchrow;
$query = "select song_id from song where (cd_ref = '$cd_ref');"; $output = $db->query($query);

# $song_id = $output->fetchrow;
while ($song_ref = $output->fetchrow) {
   $query = "insert into playsong (song_ref, playlist_ref) values ('$song_ref', '$playlistid');";
   $db->query($query);
}

$query = "select song.song_filename from playlist, playsong, song where
   (song.song_id = playsong.song_ref) &&
   (playlist.playlist_id = playsong.playlist_ref) &&
   (playlist.playlist_name = '$playlist_name');"; $output = $db->query($query);

my @x;
while ($song_fn = $output->fetchrow) {
   print $song_fn. "\n";
   push @x, $song_fn;
}

$len = $#x;
print "+++ \$len is currently $len +++\n";

$flag = 1;
while ($flag == 1) {
   $y = int(rand($#x));
   $blah = $x[$y];
   $z = 0;
   while ($z < $len) {
      if ($blah ne "used!!!") {
         system("/usr/bin/mpg123", $blah);
      }
   }
}
$x[y] = "used!!!";
last;
} else {
    $y++;
    if ($y > $len) {$y = 0;}
    $blah = $x[$y];
    $z++;
}
}
if ($z == $len) {$flag--;}

$queue = "delete from playsong
    where (playsong.playlist_ref = "$playlist_id")";
$db->query($queue);
$queue = "delete from playlist
    where (playlist.playlist_name = '$playlist_name')";
$db->query($queue);

print "Done - please scan next CD\n"}
#connect
my $db = Mysql->connect($host, $dbn, $user, $pass);

$cd_tag = shift || die "No cd_tags specified.\n"

$query = "insert into playlist (playlist_name) values ('$playlist.name');
$db->query($query);
$query = "select playlist_id from playlist
where (playlist_name = '$playlist.name');
$output = $db->query($query);
$playlist_id = $output->fetchrow;
while ($cd_tag) {
    print $cd_tag . "\n";
    $query = "select cd_id from cd where (cd_tag = '$cd_tag');
    $output = $db->query($query);
    $cd_ref = $output->fetchrow;
    $query = "select song_id from song where (cd_ref = '$cd_ref');
    $output = $db->query($query);
    # $song_id = $output->fetchrow;
    while ($song_ref = $output->fetchrow) {
        $query = "insert into playsong (song_ref, playlist_ref)
values ('$song_ref', '$playlist_id');
        $db->query($query);
    }
    $cd_tag = shift;
}
print "All tags entered.\n";

A.6 Code: wipe-db.pl

#!/usr/bin/perl -w
use Mysql;

my $db = Mysql->connect('localhost', 'cd', 'root', 'futurehome');
$db->query('delete from song where (song_id != 0)');
$db->query('delete from cd where (cdid != 0)');
$db->query('delete from playlist where (playlistjid != 0)');
$db->query('delete from playsong where (playsong-id != 0)');

%$
A.7 Data: sample CD data file

This data file format should not be taken as anything other than a prototype - it is a simple text file format, using carets as delimiters. It is used by cd-input.pl to initialize the Enhanced CD Player database. It is included for the sake of completeness.

```
Born Slippy - MUXX-01-BornSlippy.NUX.mp3
Born Slippy - MUXX - Deep Pan'02-BornSlippy.MUXX-DeepPan.mp3
Born Slippy - MUXX - Darren Price Mix'03-BornSlippy.MUXX-DarrenPriceMix.mp3
Born Slippy - MUXX - Darren Price Remix'04-BornSlippy.MUXX-DarrenPriceRemix.mp3
Born Slippy - MUXX - Short'05-BornSlippy.MUXX-Short.mp3
Dark and Long - Dark Train'09-DarkAndLong-DarkTrain.mp3
Ban Style - Alex Reece Mix'07-Banstyle-AlexReeceMix.mp3
Dark and Long'01-DarkAndLong.mp3
Mmm Skyscraper, I Love You'02-MmmSkyscraperILoveYou.mp3
Surf Boy'03-Surfboy.mp3
Spoon Man'04-Spoonman.mp3
Tongue'05-Tongue.mp3
Dirty Epic'06-DirtyEpic.mp3
Cowgirl'07-Cowgirl.mp3
River Of Bass'08-RiverOfBass.mp3
M.E.'09-ME.mp3
Signs Of Life'PF-MLOR-01-Signs_OfLife.mp3
Learning To Fly'PF-MLOR-02-Learning_To_Fly.mp3
The Dogs Of War'PF-MLOR-03-TheDogs_OfWar.mp3
One Slip'PF-MLOR-04-One_Slip.mp3
On The Turning Away'PF-MLOR-05-On_The_Turning_Away.mp3
Yet Another Movie\bBound And Around'PF-MLOR-06_YetAnother_Movie\bBound_And_Around.mp3
A New Machine (Part 1)'PF-MLOR-07\bA\bNew\bMachine,(Part,1).mp3
Terminal Frost'PF-MLOR-08\bTerminal_Frost.mp3
A New Machine (Part 2)'PF-MLOR-09\bA\bNew\bMachine,(Part,2).mp3
Sorrow'PF-MLOR-10-Sorrow.mp3
```

A.8 CGI code: http://omaha.mit.edu/cgi-bin/PML

```
#!/usr/bin/perl
use CGI qw/:standard :all :xml name tag error/;
use Mysql;
$q = new CGI;
@names = $q->param;
$tag_id = $q->param('id');
print $q->header(-type => 'text/xml');
print $q->start_xml;
# print $q->blah('TEXT');

print $q->start_xml;
# print $q->xml('TEXT');

$host = 'omaha.mit.edu';
$dbn = 'cd';
my $db = Mysql->connect($host, $dbn, 'root', 'futurehome') || print $q->error('Couldn\'t connect to database!'), $q->end_xml;
```
$query = "select cd_name from cd where (cd_tag = '$tag_id')";
$output = $db->query($query);
$cd_name = $output->fetchrow;
print $q->name($cd_name), $q->tag($tag_id);
print $q->end_xml;
Appendix B

Intelligent Shelves Code

B.1 Code: Virtual Tag Reader Network - BBox/randomize.pl

```perl
#!/usr/bin/perl
srand;
open(LOCS, "/home/DISC/Savant/BBox/locations") || die("no locations file");
open(TAGS, "/home/DISC/Savant/BBox/tags") || die("no tags file");

while(<LOCS>){
    chomp;
    $loc[($#loc+1)]=_;
    print $#loc . "\n";
}

system("rm /home/DISC/Savant/BBox/bbox.dat");
open(OUT, ">/home/DISC/Savant/BBox/bbox.dat") ||
die("Cannot open file bbox.dat for output, oops, please fix.");
print OUT "# Line anatomy: Reader_ID Tag_num\n"

while(<TAGS>){
    #base chance 75%
    $a = rand(100);
    if ($a < 75){
        #then we do something
        $b = int(rand($#loc+1));
        print "$a...$b...$#loc\n";
        print OUT "$loc[$b] 
";
    }
}
```

64
B.2 Code: timestamp.pl

#!/usr/bin/perl

use Mysql;
$usage = "programname <user> <pass> <infile>\n";

$user = shift || "root";
$pass = shift || "futurehome";
$file = shift || "/home/DISC/Savant/BBox/bbox.dat";
$host = 'omaha.mit.edu';
$dbn = 'shelves';

my $db = Mysql->connect($host, $dbn, $user, $pass);

# MAKING A NEW SNAPSHOT
$foo = get_timestamp();
my $query = "insert into snapshot (t) values ('$foo')";
$output = $db->query($query);
my $query = "select id from snapshot where (t = '$foo')";
$output = $db->query($query);

# We are going to assume that we want the most recent (i.e. highest
# numbered) snapshot. Suck.
$snap_id = 0;
while ( $bar = $output->fetchrow ) {

}
if ($bar > $snap_id) {
    $snap_id = $bar;
}

print $foo . "//" . $snap_id . "\n";

open(INF, $file);
while (<INF>) {
    next if /^#/;
    next if /^$/;
    ($loc, $tag) = split;
    #first, put the ss_tag bit in.
    my $query = "insert into ss_tag (ss_ref, tag_ref, location) values ('$snap_id', '$tag', '$loc');
    $output = $db->query($query);
    #now, put the tag placeholder in. If the tag placeholder initially points
to a s@*ty server, we're not going to fix that in this rev.
    my $query = "select id from tag where (id = '$tag')";
    $output = $db->query($query);
    if (!$output->fetchrow) {
        print "$tag not in database yet!\n";
        my $query = "insert into tag (id) values ('$tag');
        $output = $db->query($query);
    }
}

# print "$foo\n";

#WHILE WE'RE HERE... Purge old data.

$yest = get_timestamp("--date='yesterday'");

my $query = "select id, t from snapshot where ((t < '$yest') and (id % 2 != 0))";
$output = $db->query($query);
while (($outjid, $outt) = $output->fetchrow) {
    my $query = "delete from ss_tag where (tag_ref = $out-id)"
    $crap = $db->query($query);
    my $query = "delete from snapshot where (id = $out-id)"
    $crap = $db->query($query);
    print "PURGING YESTERDAY OR BEFORE: ID $outjid / TIMESTAMP $out-t\n";
}

$yest2 = get_timestamp("--date='2 days ago'")."\n";
my $query = "select id, t from snapshot where ((t < '$yest2') and (id % 4 != 0))";
$output = $db->query($query);
while ($($out_id, $out_t) = $output->fetchrow) {
    my $query = "delete from ss_tag where (tag_ref = $out_id)";
    $crap = $db->query($query);
    my $query = "delete from snapshot where (id = $out_id)";
    $crap = $db->query($query);
    print "PURGING 2 DAYS AGO OR BEFORE: ID $out_id / TIMESTAMP $out_t\n";
}

$yest3 = get_timestamp("--date=\'3 days ago\'\")."\n";
my $query = "select id, t from snapshot where (t < '$yest3')";
$output = $db->query($query);
while ($($out_id, $out_t) = $output->fetchrow) {
    my $query = "delete from ss_tag where (tag_ref = $out_id)";
    $crap = $db->query($query);
    my $query = "delete from snapshot where (id = $out_id)";
    $crap = $db->query($query);
    print "PURGING 3 DAYS AGO OR BEFORE: ID $out_id / TIMESTAMP $out_t\n";
}

#Purge tags no longer in database.
my $query = "select id from tag";
$output = $db->query($query);
while ($checking_id = $output->fetchrow) {
    #Check to see that we get a return in the ss_tag list.
    my $query = "select tag_ref from ss_tag where (tag_ref = '$checking_id')";
    $crap = $db->query($query);
    if (!$crap->fetchrow) {
        print "Tag $checking_id being purged from database.\n";
        my $query = "delete from tag where (id = '$checking_id')";
        $crap = $db->query($query);
    }
}

sub get_timestamp {
    $urk = shift || "";
    open(RDR, "/bin/date $urk --iso-8601=seconds|"");
    while (<RDR>) {
        $timestamp = $_;
        $timestamp = s/T/ /
        $timestamp = substr($timestamp,0,19);
    }
}
return $timestamp;
}

B.3 Code: test-get.pl

#!/usr/athena/bin/perl

# Create a user agent object
use LWP::UserAgent;
$ua = new LWP::UserAgent;
$ua->agent("AgentName/0.1 " . $ua->agent);

my $blah2 = shift || ""
my $blah = shift || ""

# Create a request
my $req = new HTTP::Request GET =>
'http://omaha.mit.edu/cgi-bin/Savant?timestamp='.$blah.'&query='.$blah2;
#$req->content-type('application/x-www-form-urlencoded');
#$req->content('match=www&errors=0');

# Pass request to the user agent and get a response back
my $res = $ua->request($req);

# Check the outcome of the response
if ($res->is_success) {
  print $res->content;
} else {
  print "Bad luck this time
";
}

B.4 CGI Code: Savant

#!/usr/bin/perl

use XML::XQL;
use XML::XQL::DOM;
use CGI qw/:standard :all :xml location timestamp time name tag id error/;
use Mysql;
$q = new CGI;
@names = $q->param;
$timestamp = $q->param('timestamp') || "";
$xqlparam = $q->param('query') || "";
print $q->header(-type => 'text/xml');

# print $q->blah('TEXT');

my $query_expr = $xqlparam;

#if ($query_expr eq "") {
    # print "XQL filter OFF. [\!/\*]\n\n";
#else {
    # print "XQL filter: [ $query_expr ]\n\n";
#}

# GRABBING WEB S@*& SECTION ENDS

sub reorder
{
    my @par = ();
    for (my $i = 0; $i < @order; $i++)
    {
        push @par, $_[$order[+$i]];
    }
    @par;
}

my $xqlquery;
if ($query_expr ne "") {
    eval {
        $xqlquery = new XML::XQL::Query (Expr => $query_expr, @options);
    }
}
die "$0: invalid query expression: $\@" if $\@;

my $parser = new XML::DOM::Parser;

######## ENNO BLACK MAGIC START

sub transform
{
    my $val = shift;
}
return $val unless defined $val;  # skip undef

my $type = ref($val);
return $val unless $type;    # skip scalars

if ($type eq "ARRAY")
{
    if (@$val == 0)  # empty list / undef
    {
        #??? not sure what to do here
        return "[]";
    }
    else
    {
        #??? not sure what to do here
    }
}
elsif ($val->isa ('XML::XQL::Node'))
{
    my $nodeType = $val->xql_nodeType;
    if ($nodeType == 1)  # element node
    {
        if ($E_v)
        {
            return transform ($val->xql_value);
        }
        elsif ($E_t)
        {
            return $val->xql_text ($E_recurse);
        }
        elsif ($E_r)
        {
            return $val->xql_rawText ($E_recurse);
        }
    }
    elsif ($nodeType == 2)  # attribute node
    {
        if ($A_v)
        {
            return transform ($val->xql_value);
        }
        elsif ($A_t)
        {
            return $val->xql_text;
        }
    }
elsif ($A_r)
{
    return $val->xql_rawText;
}
}
elsif ($nodeType == 3)  # text node (also CDATASection, EntityRef)
{
    if ($T_t)
    {
        return $val->xql_text;
    }
    elsif ($T_r)
    {
        return $val->xql_rawText;
    }
    }
    $val->xql_xmlString;
}
elsif ($val->isa ('XML::XQL::PrimitiveType'))
{
    #?? could add xql_normalString
    $val->xql_toString;
} else    # ???
{
    "$val";
}
}

####### ENNO BLACK MAGIC STOP

sub solveQuery
{
    my ($dom, $file) = @_; 

    my @result = $xqlquery->solve ($dom);

    # transform query results
    @result = map { transform ($_) } @result;

    # @result now stores the answer to the XQL query (passed as arg 2).
    # Right now we just print it. maybe later we do something intelligent
    # with it.
    my $foo = ""; 

    71
for (my $i = 0; $i < @result; $i++)
{
    #print $result[$i];
    #print "\n";# unless $opt_n;
    $foo = $foo . $result[$i] . "\n";
}
return $foo;

$host = 'omaha.mit.edu';
$dbn = 'shelves';
my $db = Mysql->connect($host, $dbn, 'root', 'futurehome') ||
print $q->error('Couldn\'t connect to database!'), $q->end_xml;

# hm timestamp.
# if timestamp supplied, find the argh
# get all snapshots less than that time and then take the highest UID

if ($timestamp ne "") {
    $query = "select id from snapshot where (t <= '$timestamp')";
} else {
    $query = "select id from snapshot";
}
$output = $db->query($query);
$rtsi = Relevant TimeStamp Id
$rtsi = 0;
while ($dummy = $output->fetchrow) {
    if ($dummy > $rtsi) {
        $rtsi = $dummy;
    }
}

# now we grab all ss_tags referencing the snapshot identified by $rtsi

$query = "select ss_tag.tag-ref, sstag.location from ss.tag, snapshot where ((snapshot.id = $rtsi) && (ss_tag.ssref = snapshot.id))";

# why not just select from ss_tag? because what if we f@&* up and get no
# valid snapshot, or something stupid like that.

$output = $db->query($query);
$query = "select t from snapshot where (id = $rtsi)";
spam = $db->query($query);
$spam2 = $spam->fetchrow;
#print $q->start_timestamp;

72
#print $q->time($spam2);
while (($outtag, $outloc) = $output->fetchrow) {
  # print $q->start_tag;
  # print $q->id($outtag), $q->location($outloc);
  # print $q->end_tag;
}
#print $q->end_timestamp;

# CONSTRUCT XML "FILE" HERE
$felch = "<TIMESTAMP><TIME>$spam2</TIME>";
while (($outtag, $outloc) = $output->fetchrow) {
  $felch = $felch . "<TAG><ID>$outtag</ID><LOCATION>$outloc</LOCATION></TAG>"
}
$felch = $felch . "</TIMESTAMP>";

# XQL PART - IF XQL QUERY, PARSE THE $felch STRING FOR THE QUERY
# OTHERWISE JUST RETURN THE $felch STRING
if ($xqlparam eq "") {
  print $q->xml($felch);
} else {
  my $dom = $parser->parsestring($felch);
  $felch2 = solveQuery($dom, "(webquery)");
  print $q->xml($felch2);
  #print $q->start_xml;
  #print $q->end_xml;
}
Bibliography


