THE AUSTRALIAN NATIONAL UNIVERSITY

ODBC Interface for Remote Sensor Networks

Jun Ma

University ID: u2581445

18th November 2005

A thesis submitted in part fulfillment of the degree of Master of Engineering at the Australian National University

Supervised by: Dr Ken Taylor (CSIRO)

Dr Peter Strazdins (ANU)

Declaration

I, Jun Ma, claim that the work in this Project Thesis is my own, except where due reference is made.

November 2005, Jun Ma

Acknowledgements

I would like to thank Ken Taylor and Peter Strazdins for the inspiration and continual support provided throughout the project. This field was previously unfamiliar to me, but I have found it both exciting and challenging; I would like to thank everyone at the CSIRO ICT Centre for allowing me to develop this project in such a relaxed environment.

Mum, I really appreciate the daily meals you have given me in the past couple of months, so that I can put more time and energy on my thesis.

Finally, thanks to all my family and friends that I have neglected in the past few months, now you can read about what I was doing all those long days and late nights!

Abstract

As wireless sensor networks become widely used, user-friendly interface for sending and retrieving data and easy integration with applications becomes important. These issues have largely been solved in the database domain and this project seeks to leverage these solutions by adapting sensor networks to fit the database paradigm. This is achieved by design, implementation and evaluation of an ODBC driver for remote wireless sensor networks. With such a driver, a wireless sensor network would look like a standard database table and could be used directly from the standard tools for programmatically interfacing with databases.

Through investigation of the field, the overall architecture of the system is presented and the requirements of the new interface are established. A major focus is on supporting both SQL-92 statements and TinyDB specific SQL grammar so that functionalities particular to wireless sensor networks are supported in the new interface, for example continuous sampling. Suitable software components were also selected to simplify the communication between the ODBC driver and the sensor network.

Finally, this report investigates how effectively a sensor network can be mapped onto a database paradigm given the fundamental differences and considers techniques for optimization of query latency and power consumption.

Contents

Glossary

ACA	Australian Communications Authority, the governing body that controls licensing of radio transmissions.
ANU	Australian National University.
API	Application Programming Interface.
bps	Bits per second. A bit is a binary digit - the smallest quanta of information in the digital world.
CPU	Central Processing Unit, essentially the part of a computer that does the arithmetic logic. It is commonly used to refer to a microchip as a CPU.
CSIRO	The Commonwealth Scientific and Industrial Research Organisation.
DB	Database
DBMS	Database Management System
DSN	Data Source Name
ETDEMO	Energy Transformation Demos. Developed by Distributed Energy Management and Control project group in CSIRO.
GHz	Gigahertz, a commonly used measurement of processing power.
GPRS	General Packet Radio Service, a packet switched communication protocol that utilizes the GSM mobile phone infrastructure.
GSM	Global System for Mobile Communication, the digital mobile communications infrastructure used in Australia and in many other countries around the world.
GUI	Graphical User Interface.
HVAC	Heating, Ventilation, and Air Conditioning system.
IP	Internet Protocol, a packet switched protocol which forms the basis of data transmission on the Internet.
JDBC	Java Database Connectivity.
NEMMCO	National Electrcity Market Management Company of Australia.
ODBC	Open Database Connectivity.
PHP	Hypertext Preprocessor.
RF	Radio Frequency.
RWSN	Remote Wireless Sensor Network.
SQL	Structured Query Language, a popular language used for querying and manipulating information in databases.
TCP	Transmission Control Protocol, one of the main protocols in TCP/IP networks. Whereas the IP protocol deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.
WSN	Wireless Sensor Network.
XML	Extensible Markup Language

1. Introduction

1.1. Wireless sensor networks

Wireless sensor networks (WSN) are a booming technology that many expect will one day be used everywhere on this planet. One flavor of wireless sensor network is “motes” developed at Berkeley. WSN can be used in a wide range of applications, including environmental monitoring [Mar04], surveillance [Mar04], smart buildings [Etd05], health [Asa03], transportation (traffic monitoring), and military systems. WSNs are particularly useful in these applications because deployment is easier than with other existing sensing methods. Because the hardware is both small and wireless, the applications that WSN can be applied to have great potential. As the hardware gets smaller, faster and easier to use, it becomes a more economical sensing solution and, as the demand increases, the cost to set up a WSN that covers a wide area will be driven down [Wik05]. As the potential market for WSN grows, it becomes more important that a simple, robust and easy to use interface be available, to allow ordinary users with little computing knowledge to apply WSN technology in their work.

A Berkeley WSN consists of large numbers of small-scale computers called “motes” which are equipped with limited resources like restricted CPU power, battery supply, sensing ability, communication bandwidth, etc. By making a lot of sensor nodes cooperate with each other, the WSN as a whole can provide functionality that an individual node cannot. Currently, most wireless sensors communicate via radio frequency (RF), which is limited by the strength of signal and transmission bands[1]. Many kinds of sensors are available in commercially available motes for most common measurable characteristics, including light, temperature, humidity, sound and GPS position.

The basic model of operation of WSN is similar to traditional client-server computer networks, because data processing is usually done in a centralized server. For example, a building's heating and air conditioning system (HVAC) can be monitored using temperature sensors. Hundreds of these sensors would be spread across the building, all sending their dynamic data to a central server. The server would process the data received from all sensors and consequently decide what to do with the HVAC system (turn on or off the heating or air conditioning). Since the sensors nodes are often too far from the server, many of them can not transmit directly to the server, the data is passed to other sensors which forward them until it eventually reaches the server.

The model of a typical WSN is shown in Figure 1.1. The case of Great Duck Island[Pol03] uses this model, where each sensor is represented as a sensor node. This network has a root node through which all data is passed to a basestation. The basestation is a piece of hardware that has a physical interface with the root node and another interface with the central server (usually RS232[2]). This allows all data received by the root node to be passed to the server for processing.

[1] In Australia, the ACA governs transmission licenses and public frequencies.
[2] The standard protocol used for serial connection, refer to [Str04] Make the whole sensor network looks like an

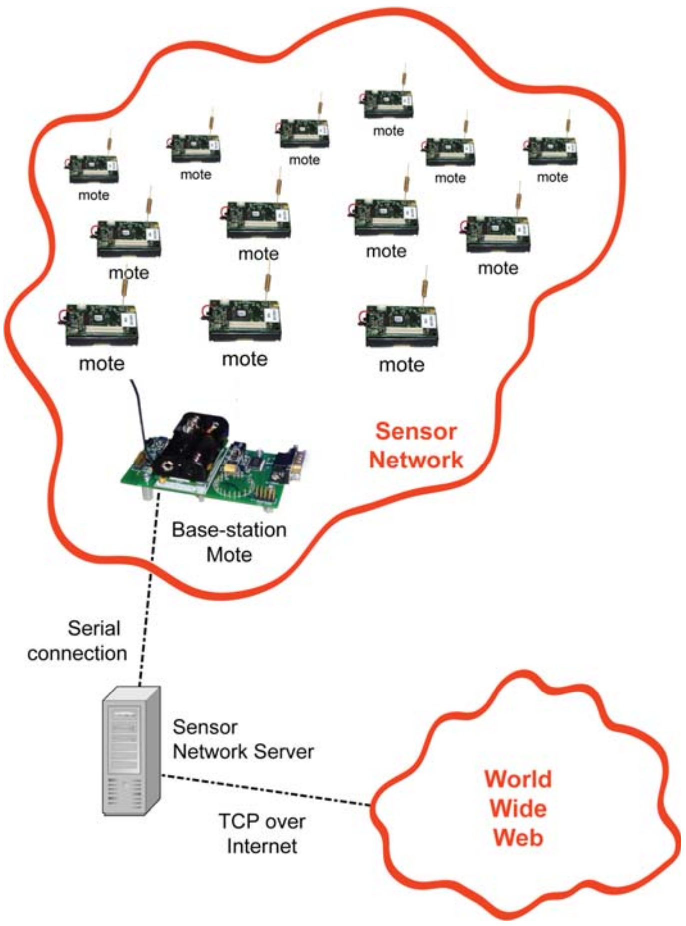


Figure 1.1 Basic WSN architecture

There are some important characteristics common to all WSNs investigated during this project, independent of the application they were developed for. The important ones to consider are:

- Each network has at least one base-station node, which must be connected to a server that has the ability to access the Internet.
- Each node is able to transmit and receive messages via radio frequency.
- Messages are passed to the base-station node using the multi-hopping communication method: packets are routed through various (intermediate) nodes in order to reach the base-station node. This also requires that all nodes have a unique identification code within the network [CO04];
- Generally the sensor nodes can be at any location, so networks can be queried according to geographical region rather than specifying a particular node [Ram03];
- Each network is extendable and flexible. With the exception of the base-station node, any node can be added or removed from the network. Normally networks are composed of a large number of sensor nodes that are densely deployed either inside the area being monitored, or very close to it;
- Sensor nodes work cooperatively, using their computing power to carry out simple computations, and then transmit only the required data [ASSC02].

1.2. A Wireless Sensor Network Application

This project is based on CSIRO Distributed Energy Management and Control Project (ETDEMO) [Etd05] which has been carried out for three years by Dr. Ken Taylor from CSIRO. During the three years, many CSIRO staff and students from ANU have contributed to this project including Kevin Mayer, Adam Wood, Les Zhang, Fahad and Hailun Tan. The project uses a distributed sensor network and smart agents to investigate new ways to facilitate energy efficiency, end user participation in the energy market and improved robustness of the electricity distribution network through the use of distributed generation and demand side energy management.

In order to provide easy connectivity to the network even when the sensor network is deployed at very remote locations, the project extended the basic WSN architecture shown in Figure 1.1 conceptually by using TinyDB as the underlying application in the sensor network. The base-station mote is connected to a device which is capable of WAN communication which is a GPRS Modem, the Ultralite [Cdcs03]. This device consists of an Atmel microprocessor and a Sony Ericsson GSM unit. By utilizing the existing GSM network, the remote sensor network can use IP over GPRS and maintain a continuous connection to the Internet.

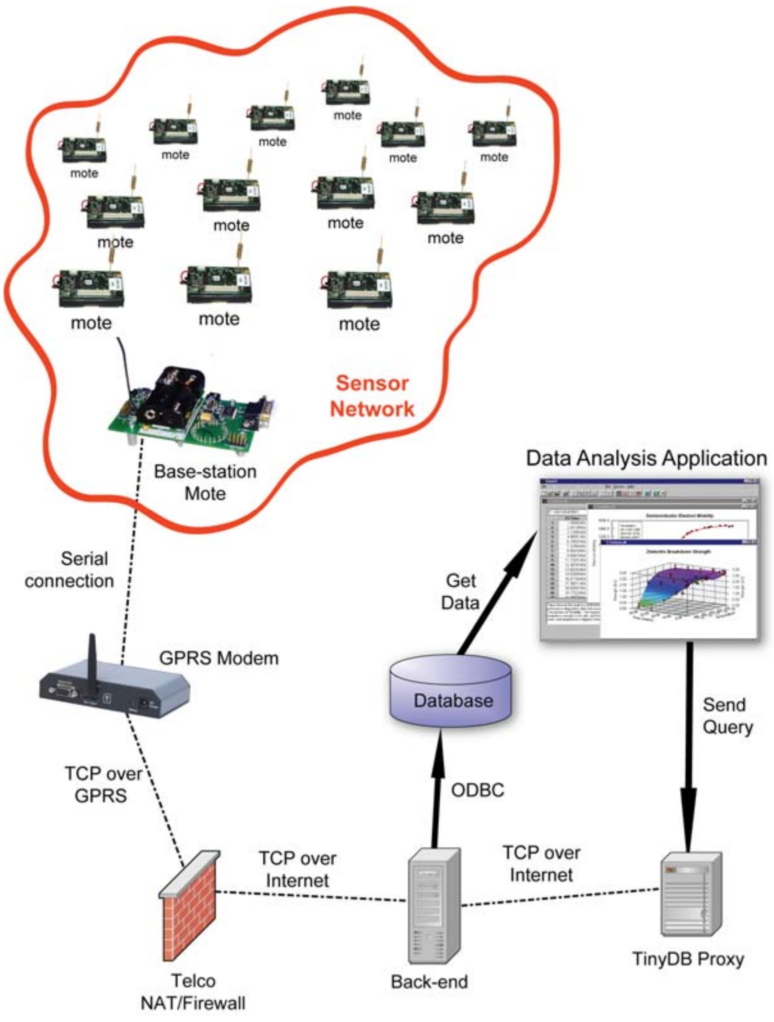
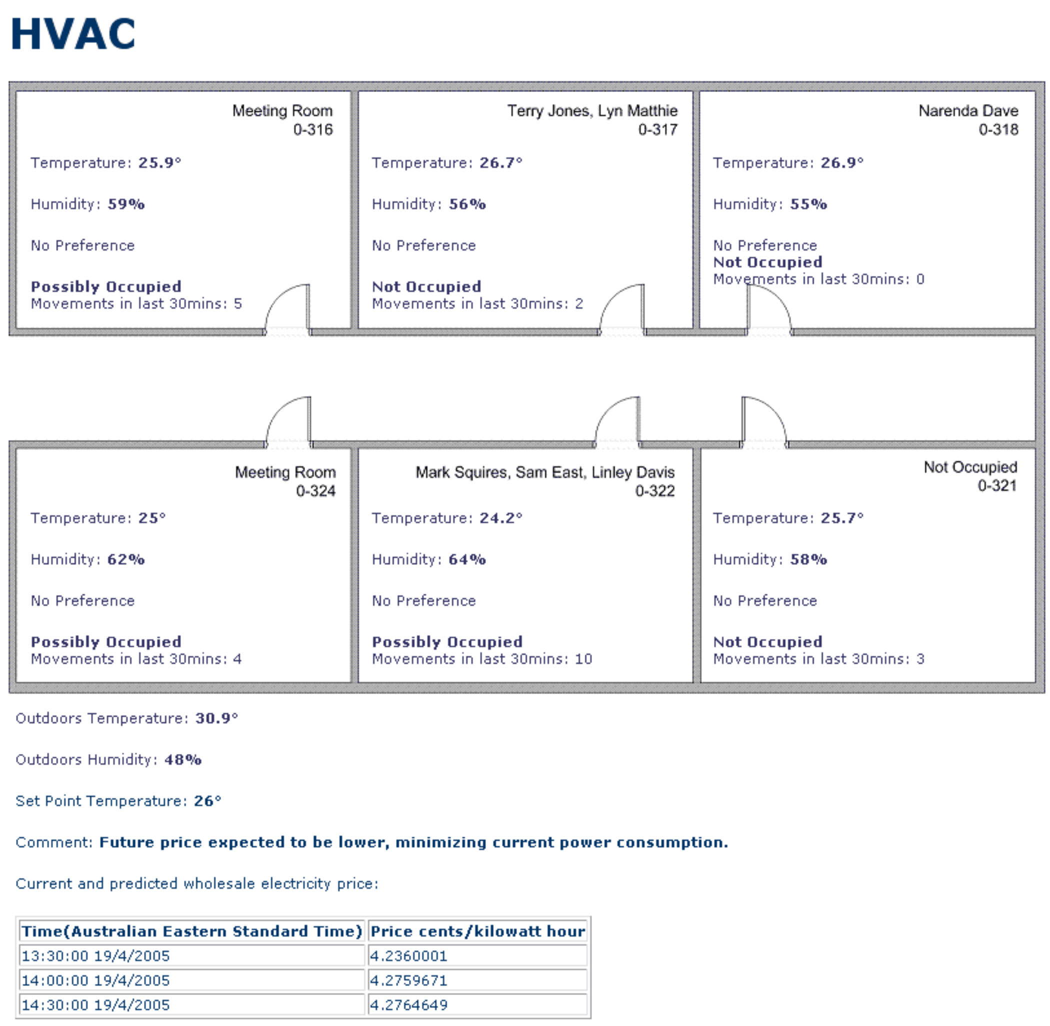


Figure 1.2 CSIRO ETDEMO system architecture

During the ETDEMO project, a number of technical demos have been created to demonstrate the potential benefit of applying sensing technology to industrial power consumption cases including HVAC[3] systems, cool rooms, fridges, wind turbines, etc. In these demo systems, a common data acquisition scheme is established so that they use the same message routing system and device interface. I call this data acquisition scheme a continuous sampling way because data is sampled periodically at a fixed sample rate and goes into a historical log table stored in a database.



HVAC

Meeting Room
0-316
Temperature: **25.9°**
Humidity: **59%**
No Preference
Possibly Occupied
Movements in last 30mins: 5

Terry Jones, Lyn Matthie
0-317
Temperature: **26.7°**
Humidity: **56%**
No Preference
Not Occupied
Movements in last 30mins: 2

Narenda Dave
0-318
Temperature: **26.9°**
Humidity: **55%**
No Preference
Not Occupied
Movements in last 30mins: 0

Meeting Room
0-324
Temperature: **25°**
Humidity: **62%**
No Preference
Possibly Occupied
Movements in last 30mins: 4

Mark Squires, Sam East, Linley Davis
0-322
Temperature: **24.2°**
Humidity: **64%**
No Preference
Possibly Occupied
Movements in last 30mins: 10

Not Occupied
0-321
Temperature: **25.7°**
Humidity: **58%**
No Preference
Not Occupied
Movements in last 30mins: 3

Outdoors Temperature: **30.9°**

Outdoors Humidity: **48%**

Set Point Temperature: **26°**

Comment: **Future price expected to be lower, minimizing current power consumption.**

Current and predicted wholesale electricity price:

Time(Australian Eastern Standard Time)	Price cents/kilowatt hour
13:30:00 19/4/2005	4.2360001
14:00:00 19/4/2005	4.2759671
14:30:00 19/4/2005	4.2764649

Figure 1.3 HVAC System Web Interface

An example of continuous sampling way for retrieving data from a sensor network is the HVAC demonstration system, ETDEMO, shown in Figure 1.3. The system is designed to monitor the temperature, humidity and occupancy of all rooms in a building using a wireless sensor network and some motion detectors. Then to control the HVAC system using an intelligent algorithm based on the sensor data it receives and the whole sale electricity price from NEMMCO[Nem05]. In Australia, electricity supply is matched to demand in the energy market managed by NEMMCO. In the market, electricity price changes constantly due to the change in demand and the price from different suppliers when they bid to sell their electricity. Figure 1.4 is a Price and Demand Graph showing the variation of electricity price in New South Wales from NEMMCO web site. As we

[3] Heating, Ventilating and Air Conditioning

see in the graph, the electricity price changes substantially over time. These price changes can be going as high as $10,000 per megawatt hour which while it occurs rarely is several hundred times the usual price.

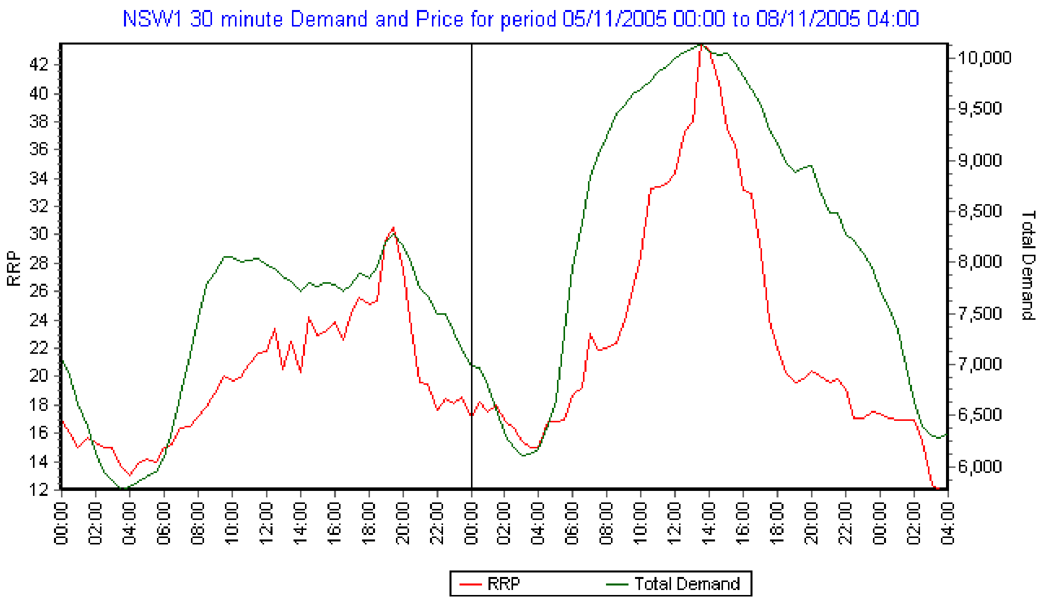


Figure 1.4 Price and Demand Graph: NSW

The idea of this system is that when a room is occupied by someone, the HVAC system will keep the temperature within a comfortable range and when the room is empty, the HVAC will try to save energy by letting the temperature become closer to the outside temperature. Also, if the future electricity price is expected to be higher, the system will try to make the room cooler than usual before the price goes up, and then turn off the compressor when the price is high until the temperature rises to the upper acceptable level, if the price is particularly high the upper acceptable level is increased at the cost of comfort. To achieve this optimization, this HVAC demo system uses motes to measure temperature and some PIR-Sensor (motion detector) to detect movements of all the rooms in the building at a sample interval of 15 minutes. Basestations get all the data and forward it to the back-end server via GPRS network over the mobile phone. Then the temperature data is stored in database from which it can be retrieved on a web page[4].

In this system, data is sampled continuously. When the application needs to get data from the sensor network, it sends an SQL query to TinyDB Proxy which translates the query into a series of TinyDB commands that motes can understand and sends them to all the sensor nodes to measure specific attributes like temperature and humidity at a specific sample rate for a certain period of time. After that, the data will periodically come back from the sensor network and is acquired by the back-end server which will write the data into a historical log table stored in a database. The application can then access the sensor data by interfacing with the database using the normally available methods e.g. JDBC, OLE-DB, ODBC, ADO Record sets, etc.

Continuous sampling has the advantage in this application that a query need only be sent once.

[4] http://etdemo.hopto.org/HVAC/hvac/default.asp

However data is only required when the application is being used and it is wasteful of sensor battery and expensive GPRS bandwidth to send this data continuously. Therefore polling is a better solution in this case.

Another problem with the existing interface is that it is non standard. Therefore we can’t take advantage of applications that automatically generate SQL database queries. For example it is possible to integrate a database table into a Microsoft Excel spreadsheet without the user becoming aware of the underlying SQL queries generated by Excel.

1.3. Project Overview

To implement a standard database interface with TinyDB, a straight forward solution is to develop an ODBC driver for the WSN. The Open Database Connectivity (ODBC) interface is an application interface (API) introduced by Microsoft that makes it possible for applications to access data from a variety of database management systems (DBMSs) [Ms05]. The ODBC interface permits maximum interoperability — an application can access data in diverse DBMSs through a single interface. Furthermore, that application will be independent of any DBMS from which it accesses data. Users of the application can add software components called drivers, which interface between an application and a specific DBMS.

The major part of this project is to develop an ODBC driver for WSN that allows normally available database tools to connect to the sensor network through an ODBC interface. For example you should be able to use Microsoft EXCEL to import data directly from the sensor network by simply selecting Data->Import External Data from the menu. With such an interface it is also possible to link the Wireless Sensor Network to SQL Server 2000 as SQL Server and some other database products can include tables from other database using an ODBC connection to a foreign database. By doing this, the sensor network can be made to appear as a standard table within such a database with the table having the interesting property of each row in the table returning a current sensor value rather than a historical record.

By the conclusion of the project, the following aims should be achieved:

- The ODBC driver should be implemented to interface elegantly to any TinyDB WSN;

The ODBC driver should provide at least the primary functionality of TinyDB: injecting a query, stopping a query and resetting the motes;

- The ODBC driver should work with the windows driver manager properly allowing the user to add, delete or modify a Data Source Name (DSN) of the ODBC driver in the driver manager.
- The sensor network should be accessible by normally available database tools like Microsoft EXCEL through ODBC interface.
- The sensor network should appears as a standard table in database with each row representing one sensor node and each column representing a different sensor or actuator on the sensor node;

- Every time a query is issued on the table which represents the WSN, the query should be translated into a TinyDB SQL query and sent to TinyDB proxy;
- After receiving the query, all nodes in the sensor network should wake up soon[5], do what the query requests and send data back to the database within a reasonably short time.

[5] TinyDB supports two power management schemes which are duty cycling and low-power listening, in both schemes, sensor wakes up periodically to sample data. So the sensor nodes can only sample data in the next sample interval.

2. Project Design

2.1. System Architecture

The architecture of this project is generally an extension to the remote sensor network architecture used in CSIRO ETDEMO shown in Figure 1.2. As shown in Figure 2.1, the new architecture has three new components which are the ODBC Driver Manager, the ODBC Driver and the XML Blaster Server. The ODBC Driver Manager is a software component provided by windows to manage the communication between the user application and the ODBC driver. The ODBC Driver is a library that provides the standard ODBC interface to user applications. And the XML Blaster Server is a message oriented middleware that I use to simplify the communication between the ODBC driver and the back-end server.

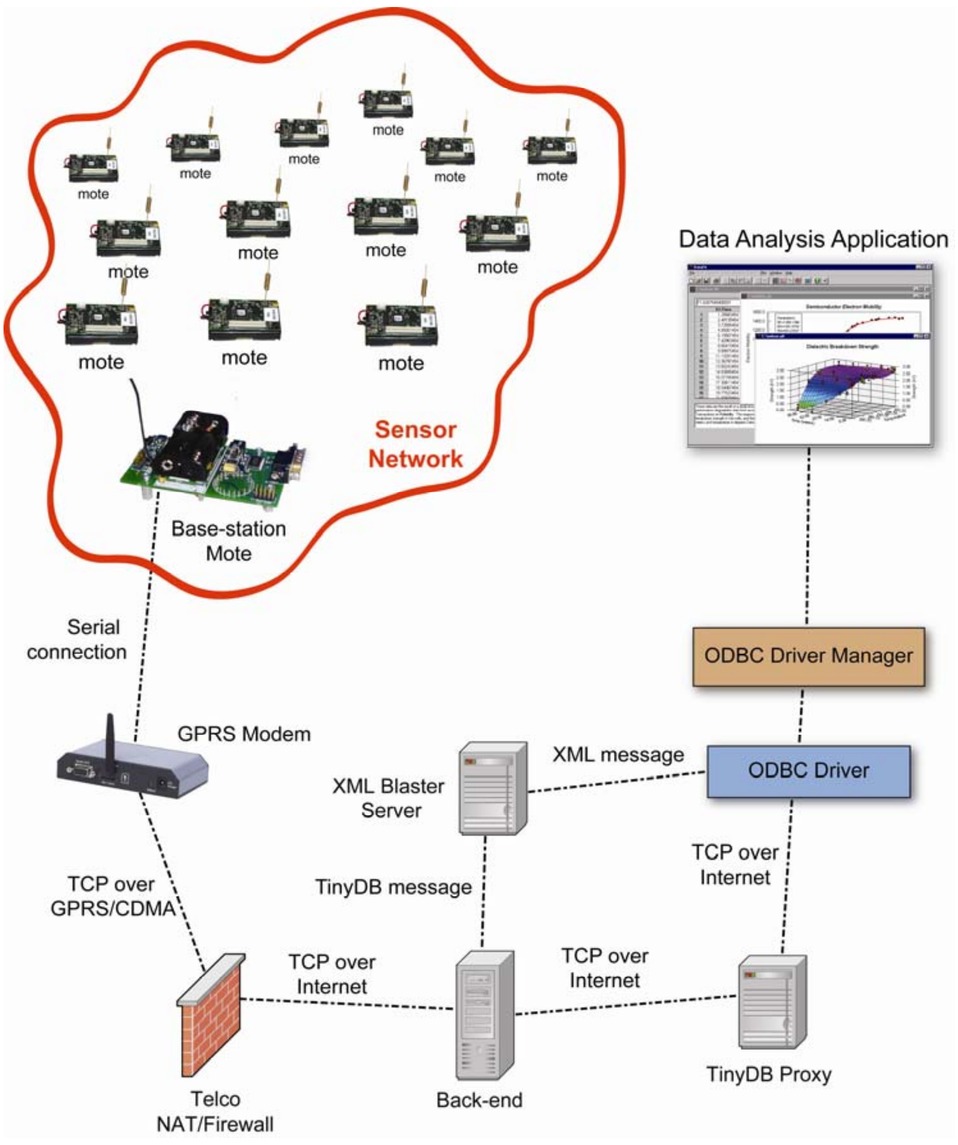


Figure 2.1 System Architecture

In the new system architecture, when a user application wants to get data from a sensor network, it does an ordinary SQL query on the system data source name (DSN) and the result it receives will be a table of the current sensor value rather than a historical record. This way of getting data from WSN is identical with the normal way of querying an ordinary database table which is familiar to ordinary users. The new way of getting data greatly improved usability, but it also introduces some problems due to the fundamental differences between a wireless sensor network and a normal database table.

One problem imposed by the new architecture is that we can’t do continuous sampling easily and frequently the historical log generated by continuous sampling is necessary. Imagine that we need to sample the temperature of a cool room every 10 minutes to draw a graph that shows how the temperature changes during one day. With the new way of getting data, we have to write a program that sends identical queries every 10 minutes. This is not as elegant as continuous sampling.

We will discuss options for continuous sampling in the later chapters.

2.2. ODBC Driver

To make the sensor network look like an ordinary table in a database, the sensor network must be transformed to a standard ODBC data source (DSN) using an ODBC driver. ODBC drivers are libraries that implement the functions in the ODBC API, in the case of the Windows platform, this library is a DLL. Each driver works with a particular DBMS; for example, an ODBC driver for Oracle cannot directly access data in an Informix DBMS. Drivers expose the capabilities of the underlying DBMSs; they are not required to implement capabilities not supported by the DBMS. For example, if the underlying DBMS does not support outer joins, then neither should the driver. The only major exception to this is that drivers for DBMSs that do not have stand-alone database engines, such as Xbase, must implement a database engine that at least supports a minimal amount of SQL. Although the underlying DBMSs have different capabilities, ODBC drivers all conform to the ODBC API which means they provide an identical interface to the user applications allowing them to access different databases without having to know the proprietary interfaces to the databases. Relying on DBMS-specific ODBC drivers, user applications can connect to several different brands of DBMSs using the same code.

In our system, specific tasks performed by the ODBC driver include:

- Connecting to and disconnecting from the XMLBlaster server and TinyDB proxy.
- Checking for function errors not checked by the Driver Manager.
- Submitting SQL statements to the TinyDB proxy for execution. The driver must modify ODBC SQL to TinyDB-specific SQL; this involves replacing “*” in the SQL statement with specific sensor attributes available in the sensor network.
- Use the XMLBlaster client to subscribe to the message published by the back-end server.
- Retrieving data from the XMLBlaster server, including converting data types as specified by the application and decoding TinyDB binary messages coming back from the sensor network.

- Resetting all the motes in the sensor network when a transaction is completed. This is because once the transaction is complete, the user application will unload the ODBC driver and the query on the sensor network should be stopped.
- Mapping TinyDB-specific errors to ODBC SQLSTATEs.
- Write windows registry information when installing a driver or adding a new DSN.
- Display dialogs to user when modifying a existing DSN.

2.3. Driver Manager

As mentioned above, user applications are designed to work with all ODBC drivers so that it can access different DBMSs using the same code. This means a user application cannot be statically linked to any drivers. Instead, they must load drivers at run time and call the functions in them through a table of function pointers. The situation becomes more complex if the application uses multiple drivers simultaneously.

Rather than forcing each application to link to a specific driver, an ODBC Driver Manager is introduced. It is a software layer between user applications and ODBC drivers which passes ODBC function calls from the user application to a specific ODBC driver. This means all user applications have to be statically linked to the driver manager all the drivers should be registered in the driver manager. Thus, the user application does not call driver functions directly. Instead, it calls a Driver Manager function with the same name and the Driver Manager calls the driver function accordingly. For example, the application calls SQLExecute in the Driver Manager and after a few error checks, the Driver Manager calls SQLExecute in the driver.

When an application wants to use a particular driver, it first requests a connection handle which is used to call a connection function (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**) in the Driver Manager and specifies the name of a particular data source or driver, such as "WSN" or "SQL Server." The Driver Manager then uses this name to search all the data source information for the driver's file name, such as ODBCWSN.dll. Once the driver’s file name is found, the Driver Manager loads the driver and stores the address of all the functions in the driver. To call an ODBC function in the driver, the application calls that function in the Driver Manager and passes the connection handle for the driver. The Driver Manager then calls the function by using the address it stored in the previous stage.

When the application has finished using the driver, it calls **SQLDisconnect** in the Driver Manager. The Driver Manager calls this function in the driver, which disconnects from the data source. However, the Driver Manager keeps the driver in memory in case the application wants to use it again. It unloads the driver only when the application frees the connection used by the driver or connects to a different driver, and there’s no other applications use the driver.

In our case, we use the Microsoft ODBC Driver Manager which is a dynamic link library (ODBC32.DLL) already installed in all Windows platform.

2.4. Linked Server

Linked Server is like an alias on local SQL server that points to an external data source. This external data source can be Access, Oracle, Excel or almost any other data system that can be accessed by OLE or ODBC--including the WSN sensor data in our case. An MS SQL linked server is similar to the MS Access feature of creating a "Link Table."

Creating a linked server can be done in two ways, by using the GUI or executing a SQL function. Using the GUI is relatively simple. To create a new linked server, open Enterprise Manager of SQLServer and go through the steps below.

- Choose Database
- Select Security
- Then choose Linked Server
- Right Click on Linked Servers
- Select New Linked Server
- Enter Server Name, and Data Source Name which is a system DSN.
- Choose Microsoft OLE DB provider for ODBC driver as Provider name.

Another way to create a linked server is to execute a function inside Query Analyzer, for example we will use the sp_AddLinkedServer function to create a linked server that points to a remote SQLServer:

```
EXEC sp_addlinkedserver
   @server = 'TEST1',
   @srvproduct = 'SQLServer OLEDB Provider',
   @provider = 'SQLOLEDB',
   @datasrc = 'InfoNet'
```

@server is the Link name. @datasrc is the remote SQL server.

After creating the linked server the remote data source (WSN) can be accessed by calling openquery() function in a SQL query. For example, the following query will select everything from a table called JunMa in a linked server called WSN.

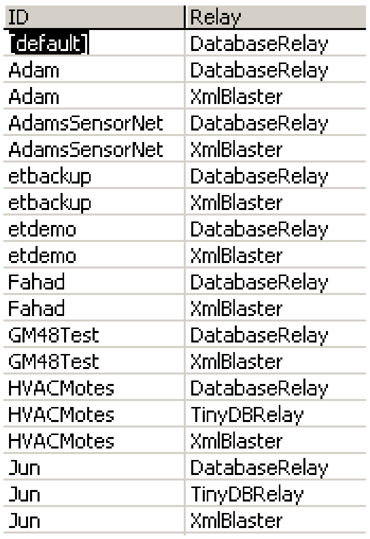
```
SELECT * FROM OPENQUERY(WSN, ‘select nodeid,temp from Jun’)
```

2.5. XMLBlaster

One important question in this project is that how the ODBC driver gets data from the back-end server which handles data coming from the sensor network. This requires a message queue and some kind of fault tolerance mechanism; also the message should be in an understandable format for ease of processing. These services are provided by a good message oriented middle ware and the open sourced XMLBlaster has been chosen for this purpose.

XMLBlaster is java based middleware developed by xmlblaster.org that allows distributed clients to send XML messages to each other using CORBA and various other protocols like RMI or XmlRpc. XML Messages may contain anything, GIF images, Java objects, Python scripts, XML data, a word document, plain text - just anything. XMLBlaster server uses publish/subscribe to organize the message exchange process, so that a client can subscribe to a certain publisher and whenever a publisher sends a message the subscriber will receive it. Most importantly, XMLBlaster is available for free which makes it a good choice to glue together our ODBC driver and the back-end end to form a distributed client/server application.

In this system, XMLBlaster is used to send query results back in XML format to the ODBC driver as follows. When a query result comes back to the back-end server, the back-end server checks which device it is from and does a database query on the DeviceConfigurationRelay table as shown in Figure2.2, if the device name is set to be XmlBlaster Relay, the back-end will then publish the XML message to XMLBlaster server. The XMLBlaster server gets the data and forwards it to all the clients that subscribe to it. As the ODBC driver is designed to contain an XMLBlaster client module and it subscribes to the query result data, it shall receive the query result from XMLBlaster server.



ID	Relay
[default]	DatabaseRelay
Adam	DatabaseRelay
Adam	XmlBlaster
AdamsSensorNet	DatabaseRelay
AdamsSensorNet	XmlBlaster
etbackup	DatabaseRelay
etbackup	XmlBlaster
etdemo	DatabaseRelay
etdemo	XmlBlaster
Fahad	DatabaseRelay
Fahad	XmlBlaster
GM48Test	DatabaseRelay
GM48Test	XmlBlaster
HVACMotes	DatabaseRelay
HVACMotes	TinyDBRelay
HVACMotes	XmlBlaster
Jun	DatabaseRelay
Jun	TinyDBRelay
Jun	XmlBlaster

Figure 2.2 DeviceConfigurationRelay table

Another problem that should be mentioned is that sometimes when more than one device publishes messages to the XML Blaster Server, the ODBC driver will receive some messages that it can not decode. This requires some message filtering mechanism to exist in the ODBC driver to filter out those irrelevant messages. This can be done either in the XML Blaster server by setting a separate message queue for each device or in the ODBC driver by parsing the XML message and selecting messages from a specific device name.

2.6.Back-end Server

As mentioned in chapter 2.5, data from the sensor network has to be sent to the XMLBlaster server through a back-end server. The back-end server is, essentially, a routing system to allow simple devices behind a NAT to communicate with the internet. As shown in Figure 2.3, it has a simple routing structure with many interfaces including an XMLBlaster interface connect to external systems via sockets, database connections, etc. The interface is responsible for maintaining the connection. The back-end routes information from one interface to another.

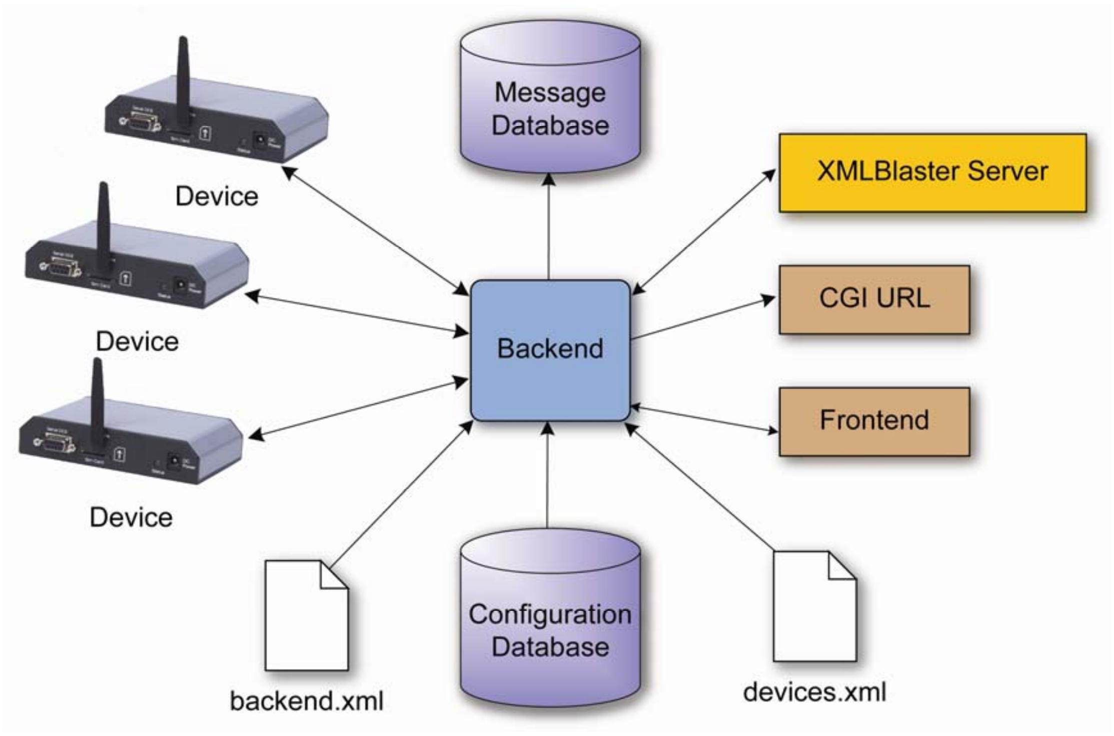


Figure 2.3 Back-end Architecture

There are two configuration files and one configuration database used in the back-end. The backend.xml in is a configuration file that describes what interfaces to use and how to translating the internal, object-based representation of information into various formats suitable for that interface. The devices.xml is a configuration file that contains information on how specific, named devices should be configured. The configuration database also contains information on how specific, named devices should be configured.

There are several device interfaces in the back-end to connect to various external devices, eg. Ultralite devices. For each device, two socket connections are created; a command channel for login, program download and other control messages and a data channel for data messages. In our case, we use the device interfaces to connect to the Ultralite GPRS modem so that we can get data from the sensor network.

The back-end also has a database interface which connects to the external message database via a direct JDBC database connection. With such an interface, the messages from devices can be stored into a database table. This interface is used in the previous ETDEMO system to store all the data from the sensor network into the database.

In our case, the most important interface is the interface to the XMLBlaster middleware that allows messages from devices to be routed into the XMLBlaster server and requests from clients to come via XMLBlaster. That's how the ODBC driver receives the query result back from sensor network.

A web CGI connection is also introduced in the back-end. It allows messages from devices to be routed to a suitable URL.

The front-end interface in the back-end allows requests and commands to be sent to the back-end or devices and the results returned. A simple socket-based request-reply protocol is used. A front-end listener on a well-known port listens for connections,

2.7. GPRS Modem

The GPRS Modem is used to forward TinyDB messages from the sensor network to the back-end server and vice versa (based on TinyDB proxy). In this project, we uses a GPRS modem distributed by Call Direct[Cdcs03] specifically designed to interface with RS232 (serial) compatible devices.

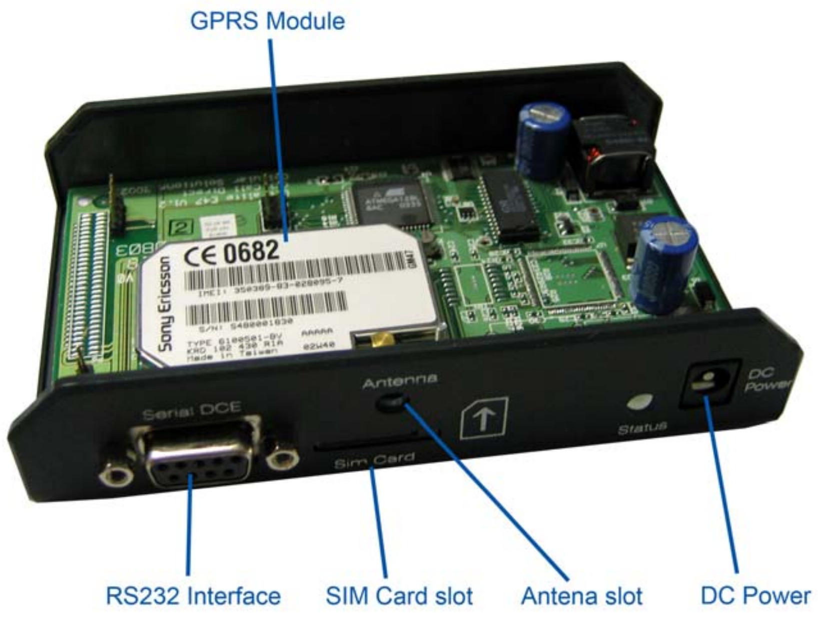


Figure 2.4 Ultralite GPRS Modem

A C program called TinyOSBridge, originally developed by Kevin Mayer is used as the program on the Ultralite. The program converts between serial packets (for the RS232 interface) and PPP/IP packets (for the GPRS interface). And when forwarding TinyDB messages from the RS232 interface to the GPRS interface, it reads from the RS232 interface in binary format, and encodes them into text format using Base64 encoding which is a method of encoding arbitrary binary data as ASCII text. Base64 encoding takes three bytes, each consisting of eight bits, and represents them as four printable characters in the ASCII standard, the same encoding method used for sending binary data in e-mail. Then the encoded message is integrated into an XML message and sent over the GPRS network to the back-end server.

PPP/IP packets from the backend server have the format:

```
   struct TOS_Msg {
      uint16_t addr;
      uint8_t type;
      uint8_t group;
      uint8_t length;
      int8_t data[length];
      uint16_t crc;
   }
```

The serial packets have the format:

```
 0x7E    byte[1]    byte[1]    byte[length]          byte[2]    0x7E
 flag    protocol    seq. #    TOS_Msg (payload)     FCS        flag
```

Features of the XML message:

```
        <message>
                <timestamp>
                </timestamp>
                <address>
                </address>
                <contents>
                </contents>
        </message>
```

3. ODBC driver development

3.1. Introduction

As mentioned in the last chapter, to make the sensor network look like an ordinary table in a database, the sensor network must be transformed to an ODBC data source using a driver. ODBC drivers are libraries that implement the functions in the ODBC API, in case of the Windows platform, the driver is a DLL.

The file formats for DLLs are same as for Windows EXE files. As with EXEs, DLLs can contain code, data, and resources. DLLs can be created by lots of compilers including Visual C++, DELPHI and Visual Basic. For my project, Visual C++ has been chosen to be the development environment because the ODBC interface while language independent was originally designed for use with the C programming language [Msdn05] and Microsoft has provided a good SDK for ODBC called Microsoft Data Access SDK which contains the full specification of ODBC API, lots of useful documentation and test tools.

3.2. Export Functions

The first thing to look at when developing a driver is how to export functions. Unlike ordinary programs that constitute a process, a driver is designed to provide a collection of functions that can be loaded into a process and invoked at run time. This will require some mechanism to export a function from the driver so that other programs can call the function. In Visual C++, Microsoft introduced some extensions for specifying export functions. In the 16-bit compiler version of Visual C++, __export allows the compiler to generate the export names automatically and place them in a .LIB file. This .LIB file could then be used just like a static .LIB to link with a DLL.

In the 32-bit version of Visual C++, functions, classes, or class member functions can be exported from a DLL using the __declspec(dllexport) keyword. For example, to export a function, the __declspec(dllexport) keyword should appear to the left of the function definition like this:

```
        __declspec(dllexport) void __cdecl Function1(void);
```

Adding a __declspec(dllexport) is easy and convenient in most cases when trying to export C++ function names. **But, an ODBC driver CAN NOT use this way to export functions.** This is because the compiler uses name decoration when creating the .LIB and .DLL files. To make it simple, the names of exported functions are changed during the compilation. See Figure 3.1 to get a rough idea about what name decoration is.

EXPORTED FUNCTION	ORDINAL	DECORATED NAME
Prod	1	?Prod@@YAJJJ@Z
Sum	2	?Sum@@YAJJJ@Z
CMyClass::CMyClass()	3	??0CMyClass@@QAE@XZ
CMyClass::~CMyClass()	4	??1CMyClass@@QAE@XZ
CMyClass::operator=	5	??4CMyClass@@QAEAAV0@ABV0@@Z
CMyClass::SAbout()	6	?SAbout@CMyClass@@QAEXXZ
CMyClass::SHowdy()	7	?SHowdy@CMyClass@@QAEXXZ

Figure 3.1 Function Name Decoration

Name decoration is good when the DLL and dependent .EXE files are generated by the same compiler because it provides more information about an exported function like how many arguments are needed and their types. But for a library like ODBC driver, name decoration will make it useless because there is no standard specification for name decoration, so the name of an exported function may change between compilers and even different compiler versions. If we use __declspec(dllexport) , most user applications will not be able to call any functions in the driver because they can't understand the name decoration.

To make the ODBC driver work with any user applications, the exported functions should have their original names rather than decorated names. This kind of export directives can be made only in a .DEF file, and there is no way to export functions using original names without a .DEF file.

A module-definition (.DEF) file is a text file containing one or more module statements that describe various attributes of a DLL. A minimal .DEF file must contain the following module-definition statements:

- The first statement in the file must be the LIBRARY statement. This statement identifies the .DEF file as belonging to a DLL. The LIBRARY statement is followed by the name of the DLL. The linker places this name in the DLL's import library.

- The EXPORTS statement lists the names and, optionally, the ordinal values of the functions exported by the DLL. An ordinal value can be assigned to a function by following the function's name with an at sign (@) and a number. When specifying ordinal values, they must be in the range 1 through N, where N is the number of functions exported by the DLL.

For example, an ODBC driver DEF might look like the following:

```
LIBRARY ODBCWSN
EXPORTS
;SQLAllocConnect @1
;SQLAllocEnv @2
;SQLAllocStmt @3
SQLBindCol @4
SQLCancel @5
…….
```

When compiling, the DEF file should be included in the source directory and added to the "module definition file" option in the linker. For more details, please refer to 3.7.2. Compiling the ODBC Driver.

3.3. Driver Architecture

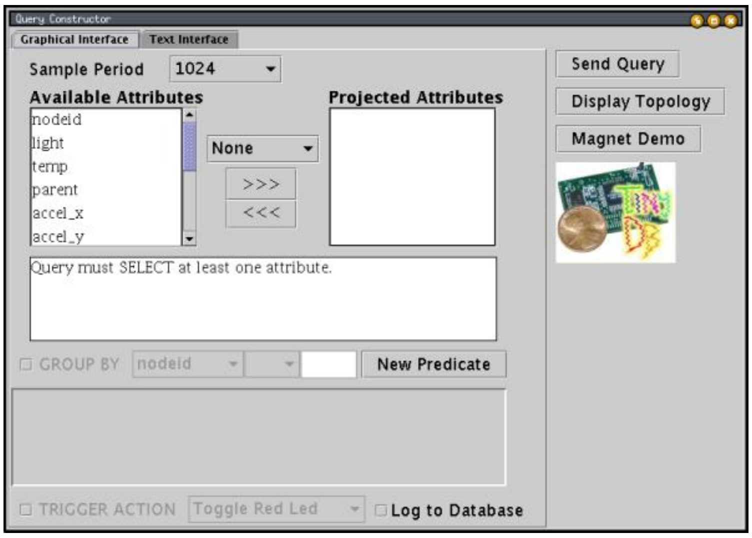
Generally there're two kinds of ODBC drivers, one is File-based drivers which are used with data sources such as dBASE that do not provide a stand-alone database engine for the driver to use. These drivers access the physical data directly and must implement a database engine to process SQL statements. As a standard practice, the database engines in file-based drivers implement the subset of ODBC SQL defined by the minimum SQL conformance level; for a list of the SQL statements in this conformance level, see Appendix A: SQL Minimum Grammar.

The other kind of ODBC driver is a DBMS-based driver which is used with data sources such as Oracle or SQL Server that provide a stand-alone database engine for the driver to use. These drivers access the physical data through the stand-alone engine; that is, they submit SQL statements to and retrieve results from the engine.

The ODBC driver for WSN in this project is based on TinyDB, so it is more like a DBMS-based driver that uses an existing database engine. But because TinyDB is not a database engine, the driver architecture has to be modified so that it can appear to the user application like a normal database.

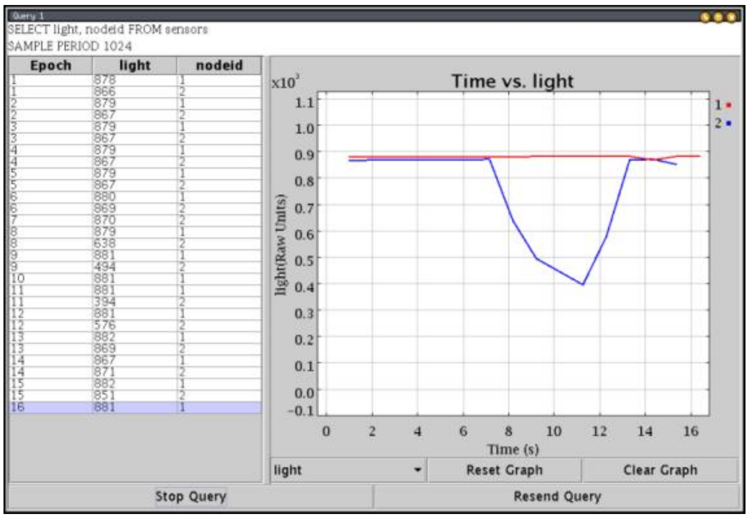
Unlike normally available database engines, TinyDB is just a simple query processing system for extracting information from a network of motes written in java. As you can see in its interface in Figure3.1 and Figure 3.2, there's no such thing as users, authentication, databases and tables in TinyDB. The only thing that makes it look like a database engine is the SQL statement, but unlike normal SQL statements, the SQL statement in TinyDB is only used for extracting data from the sensor network which means SQL statements like INSERT or CREATE don't make sense and the only useful statement is SELECT.

Another difference between TinyDB and normal available database engines is that, a very normal query like "SELECT * FROM wsn" can not be executed in TinyDB because that statement means "SELECT all the attributes from the sensor network" but TinyDB itself does not know what attributes are available. As you can see in the query construct interface shown in Figure 3.1, the available attributes of the sensor network must be specified by the user, this is mainly because the available attributes of the sensor network is determined by the sensor board you installed on the motes, for example, when you installed a temperature sensor on the motes, you should be able to do a “SELECT temp FROM wsn” and when you installed a light sensor on the mote, you can do “SELECT light FROM wsn”. So, in the ODBC driver for WSN, a user should be able to specify what attributes are available in the sensor network and query to any other attributes should result an error.


Sample Period: 1024

Query must SELECT at least one attribute.

Figure 3.1 TinyDB Interface.


SELECT light, nodeid FROM sensors
SAMPLE PERIOD 1024

Epoch	light	nodeid
1	878	1
1	866	2
2	879	1
2	867	2
3	879	1
3	867	2
4	879	1
4	867	2
5	879	1
5	867	2
6	880	1
6	869	2
7	870	2
8	879	1
8	638	2
9	881	1
9	494	2
10	881	1
11	881	1
11	394	2
12	881	1
12	576	2
13	882	1
13	869	2
14	867	1
14	871	2
15	882	1
15	851	2
16	881	1

Figure 3.2 TinyDB Query Result Interface.

As mentioned above, an ODBC driver provides a collection of functions that can be called by user application. In other words, a user application will call functions in the driver in a certain sequence to get data from the data source. To get the idea of what major functions in an ODBC driver are getting called for processing a query, please refer to a basic application steps chart made by Microsoft at [Msd05]:

Because TinyDB doesn't support SQL statements like UPDATE or CREATE, I modified the chart of the basic application steps to make it suit the special case we have as you can see in Figure 3.3. This chart gives a rough but simple basic process of how the ODBC driver for WSN works. In the beginning of the driver development process, I mainly focused on the functions called in the basic process and sooner I found out more functions are called in the ODBC driver by certain applications.

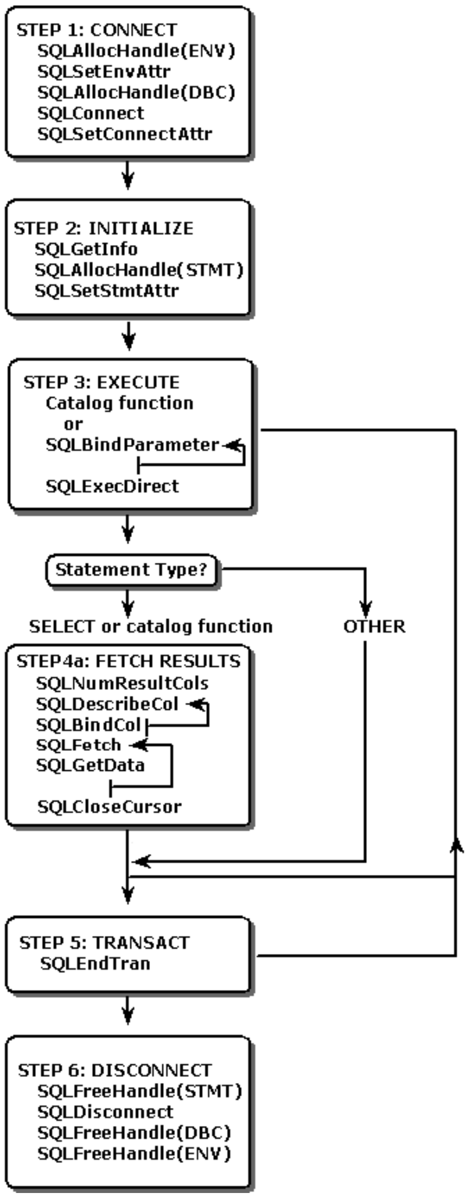


Figure 3.3 Major ODBC functions called by applications.

3.4. Conformance Levels

Conformance level informs an application what features are available to it from the driver and lets the application determine at run time what ODBC capabilities and what SQL grammar the driver and each data source supports. Because of the fundamental differences between a database table and a sensor network, choosing a proper conformance level can save lots of time and energy because the full conformance level provides too much functionality that few databases implement them all. In other words, lots of functionalities in normal databases do not exist in a sensor network, like inserting a row into a wireless sensor network is meaningless. To help normally available ODBC applications discover the driver capabilities and work properly with the ODBC driver, two areas of conformance must be considered: the ODBC interface and SQL grammar.

3.4.1. Interface Conformance Levels

Interface conformance levels are generally about what features are supported in the ODBC interface. It does not always divide neatly into support for a specific list of ODBC functions, but specifies supported features as listed in Appendix B: Core Level Conformance. To provide support for a feature, a driver must support some or all forms of calls to certain ODBC functions.

The application discovers a driver's interface conformance level by connecting to a data source and calling SQLGetInfo with the SQL_ODBC_INTERFACE_CONFORMANCE option. As drivers are free to implement features beyond the level to which they claim complete conformance, another function is needed to discover any such additional capabilities. ODBC has solved this problem by introducing a function SQLGetFunctions to determine which ODBC functions are present.

For example, when Microsoft EXCEL connects to an ODBC driver, it will call SQLGetFunctions() with the SQL_API_ODBC3_ALL_FUNCTIONS option. The return value is an array that is treated by the Driver Manager as a 4,000-bit bitmap that can be used to determine whether an ODBC 3.*x* or earlier function is supported. By using this array, we can inform EXCEL that certain functions are not available in the ODBC driver.

3.4.2. SQL Conformance Levels

SQL Conformance Levels are generally about what subset of SQL-92 grammar is supported by a driver. An application gets the SQL Conformance Level by calling SQLGetInfo with the SQL_SQL_CONFORMANCE information type. This indicates whether the driver conforms to the Entry, FIPS Transitional, Intermediate, or Full levels defined in SQL-92.

As specified in [Ms05], all ODBC drivers must support the minimum SQL grammar described in Appendix A: SQL Minimum Grammar. This grammar is a subset of the Entry level of SQL-92. Drivers may support additional SQL and be conformant to the SQL-92 Entry, Intermediate, or Full level, or to the FIPS 127-2 Transitional level. Drivers that comply to a given level of SQL-92 or FIPS 127-2 can support additional features in any of the higher levels yet not be fully conformant to that level. To determine whether a feature is supported, an application should call SQLGetInfo with the appropriate information type. The conformance level of an SQL feature is described in the corresponding information type.

To simplify the driver development, the driver aims at minimal conformance level. Also because some SQL statements do not make sense in the case of wireless sensor networks, only a subset of the SQL minimal grammar is supported. For example, at the moment, only the SELECT statement which selects specific attributes or “*” is supported in the driver.

3.5. Programming Considerations

An ODBC driver must be fully thread-safe and it must not require thread affinity. That is, the driver must be able to handle an ODBC call from any thread at any time. For example, it must be possible to connect to the database from a call on one thread, to use the database connection from another thread, and to disconnect from the database from another thread. This is essential because the ODBC driver manager opens and closes database connections from a system thread, while application components may call the database from any of a number of application threads.
If an ODBC driver is not thread-safe, a memory access violation may occur. This is most likely to occur after a database connection has been inactive for 60 seconds and the driver manager attempts to close the database connection.[Ref05]

Although a DBMS-based driver can be easily implemented by translating ODBC calls to native API calls, this results in a slower driver. A better way to implement a DBMS-based driver is to use the underlying data stream protocol, which is usually what the native API does. For example, a SQL Server driver should use TDS (the data stream protocol for SQL Server) rather than DB Library (the native API for SQL Server). An exception to this rule is when ODBC is the native API. For example, Watcom SQL is a stand-alone engine that resides on the same machine as the application and is loaded directly as the driver [Ref05].

In our case, the native API of TinyDB proxy is by using XMLBlaster and the TinyDB message binary format. So instead of getting query results back from TinyDB proxy, I have chosen to talk directly to XMLBlaster when getting data back from the sensor network and decode the TinyDB message within the driver. By doing this, the driver retrieves results faster and we can get access to the raw TinyDB message which contains some information you can not get from TinyDB proxy like how many fields are available in the query result.

The query statement can also be sent directly to the sensor network by our ODBC driver which is a more straight forward way, but because the TinyDB proxy has provided a simple interface for sending queries, I have chosen not to bypass TinyDB proxy, the driver simply opens a socket and write the SQL statement string to port 1320 of the TinyDB proxy server and the statement will be converted into a TinyDB message and sent to the sensor network. The string sent to TinyDB proxy must be terminated with a new line character ‘\n’ instead of a usual null terminator ‘\0’. This is a common problem with a C client application talking to a java server because the TinyDB proxy is a java server and java uses ‘\n’ to determine the end of the string.

After TinyDB has received the SQL statement, it will try to send it to the sensor network and send back a message indicating whether the query is sent successfully or not. The message consists of at most two characters, the first char actor indicates the query is a success one (char1=’2’) or not (char1=’1’), the second char provides an error code when the query is not successfully sent. The error message is as follows:

Char value	Error message
‘1’	An internal error has occured.
‘2’	You entered an invalid SQL string.
‘3’	The server could not be contacted.
‘4’	The device you are trying to query is not currently online.
‘5’	The maximum number of connections to the Server has been exceeded.

3.6. Compilation Notes

3.6.1. Compiling XMLBlaster

When compiling XMLBlaster client, a #define _WINDOWS must exist before including any header files or the compiler will try to find some header files like “stdint.h” which do not exist in windows.

The XMLBlaster client is linked to four dynamic link libraries, thus the four DLL files must be copied to the same directory as the ODBC driver, or WINDOWS\SYSTEM32 directory or any directory in the PATH environmental variable so that the ODBC driver is able to find them. The five DLL files are:

- xmlBlasterClient.dll
- xerces-c_2_6.dll
- xerces-depdom_2_6.dll
- xmlBlasterClientC.dll
- pthreadVC2.dll

3.6.2. Compiling the ODBC Driver

This section describes how to build the ODBC Driver for WSN (ODBCWSN.dll). Microsoft Visual C++.NET version 7.0 or higher is required. Other compilers may work but have not been tested.

1. Create a new project workspace with the type DLL. For the name, type in the name "ODBCWSN".

2. The above step creates the directory “ODBCWSN” under the “\<Visual C++ top level directory>\projects” path to hold the source files. (for example, \ My Documents\Visual Studio Projects \ODBCWSN). Now, either unzip the source code release into this directory or just copy all the files into this directory.

3. Insert all of the source files (*.cpp, *.h, *.rc, *.def) into the Visual project using the "Insert files into project" command. You may have to do 2 inserts -- the first to get the 'cpp' and header files, and the second to get the def file. Don't forget the .def file since it is an important part of the release. You can even insert ".txt" files into the projects -- they will do nothing.

4. Add the “wsock32.lib” and “xmlBlasterClient.lib” library to the end of the list of libraries for linking using the Build settings menu.

5. Add the “odbcwsn.def” to the module definition file in the link settings.
6. Select the type of build on the toolbar (i.e., Release or Debug). This is one of the useful features of the visual c++ environment in that you can browse the entire project if you build the “Debug” release. For release purposes however, select “Release” build.

7. Build the dll by selecting Build from the build menu.

8. When complete, the “ODBCWSN.dll” file is under the “Release” subdirectory.

You can use the dumpbin tool to check all the exported functions in the dll file by typing this command:

```
dumpbin ODBCWSN.dll /EXPROTS /OUT:ODBCWSN.def
```

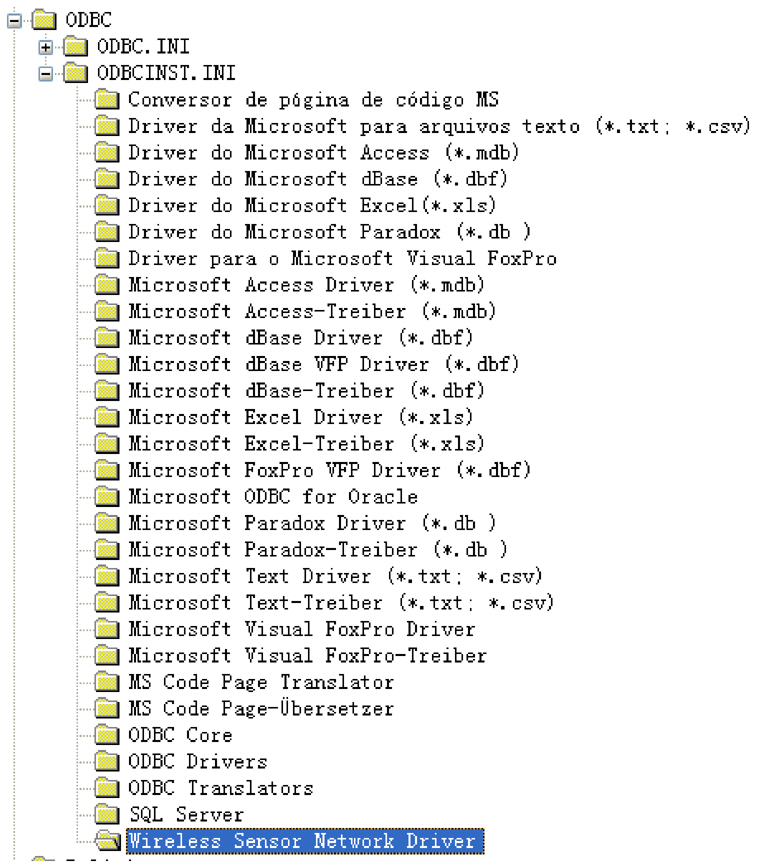
The exported function names in the def file should look like this:

```
202    0    00017DBB    ConfigDSN
200    1    00017276    ConfigDriver
201    2    00017267    DllMain
 80    3    00017866    SQLAllocHandle
  4    4    000176C7    SQLBindCol
 81    5    00017956    SQLBindParam
 72    6    00017A14    SQLBindParameter
 55    7    0001705A    SQLBrowseConnect
100    8    00017915    SQLBulkOperations
  5    9    00017EA6    SQLCancel
```

If the function names are not right, make sure the “odbcwsn.def” is included in the source folder and added to the link settings. The def file specifies the export functions and must be added to the linker setting.

3.7. Driver Installation

To make the ODBC driver appear in the Microsoft ODBC Driver Manager, we need to install it appropriately. On windows platform, all the ODBC drivers are dynamic link library (.dll) files registered in the Windows system registry. To install a driver we need to write the path of the ODBCWSN.dll and various driver information discussed below into the windows registry accordingly.



```
ODBC
  ODBC.INI
  ODBCINST.INI
    Conversor de página de código MS
    Driver da Microsoft para arquivos texto (*.txt; *.csv)
    Driver do Microsoft Access (*.mdb)
    Driver do Microsoft dBase (*.dbf)
    Driver do Microsoft Excel(*.xls)
    Driver do Microsoft Paradox (*.db )
    Driver para o Microsoft Visual FoxPro
    Microsoft Access Driver (*.mdb)
    Microsoft Access-Treiber (*.mdb)
    Microsoft dBase Driver (*.dbf)
    Microsoft dBase VFP Driver (*.dbf)
    Microsoft dBase-Treiber (*.dbf)
    Microsoft Excel Driver (*.xls)
    Microsoft Excel-Treiber (*.xls)
    Microsoft FoxPro VFP Driver (*.dbf)
    Microsoft ODBC for Oracle
    Microsoft Paradox Driver (*.db )
    Microsoft Paradox-Treiber (*.db )
    Microsoft Text Driver (*.txt; *.csv)
    Microsoft Text-Treiber (*.txt; *.csv)
    Microsoft Visual FoxPro Driver
    Microsoft Visual FoxPro-Treiber
    MS Code Page Translator
    MS Code Page-Übersetzer
    ODBC Core
    ODBC Drivers
    ODBC Translators
    SQL Server
    Wireless Sensor Network Driver
```

Figure 3.4 ODBC drivers information in registry

All the installed ODBC drivers are registered in the windows registry as you can see in Figure 2.3 which is a snapshot of windows registry editor at:

HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI

To properly install an ODBC driver, we need to have a number of keys in the registry, to find out what info we need to register a driver, let’s look at the registry of the Wireless Sensor Network Driver:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\
Wireless Sensor Network Driver ]
"APILevel"="1"
"ConnectFunctions"="YYY"
```

```
"DriverODBCVer"="03.00"
"FileUsage"="0"
"SQLLevel"="0"
"Driver"="C:\\WINDOWS\\system32\\ODBCWSN.dll"
"UsageCount"=dword:00000002
"Setup"="C:\\WINDOWS\\system32\\ODBCWSN.dll"
"CPTimeout"=""
```

Generally, it tells the driver manager where the driver library file is located and various features supported by this driver. A brief explanation is as follows:

- UsageCount: The usage count of the driver, suppose the ODBC driver have been used to setup three different DSNs The value under the ODBC Core subkey would be:

  UsageCount : REG_DWORD : 0x3
- Driver: Full paths of the driver DLL.
- Setup: Full paths of the driver setup DLL.
- APILevel: A number indicating the ODBC interface conformance level supported by the driver: 0 = None, 1 = Level 1 supported, 2 = Level 2 supported. This must be the same as the value returned for the SQL_ODBC_INTERFACE_CONFORMANCE option in SQLGetInfo.
- ConnectFunctions: A three-character string indicating whether the driver supports SQLConnect, SQLDriverConnect, and SQLBrowseConnect. If the driver supports SQLConnect, the first character is "Y"; otherwise, it is "N". If the driver supports SQLDriverConnect, the second character is "Y"; otherwise, it is "N". If the driver supports SQLBrowseConnect, the third character is "Y"; otherwise, it is "N". For example, if a driver supports SQLConnect and SQLDriverConnect but not SQLBrowseConnect, the three-character string is "YYN".
- DriverODBCVer: A character string with the version of ODBC that the driver supports. The version is of the form nn.nn, where the first two digits are the major version and the next two digits are the minor version. For the version of ODBC described in this manual, the driver must return "03.00". This must be the same as the value returned for the SQL_DRIVER_ODBC_VER option in SQLGetInfo.
- SQLLevel: A number indicating the SQL-92 grammar supported by the driver: 0 = SQL-92 Entry, 1 = FIPS127-2 Transitional, 2 = SQL-92 Intermediate, 3 = SQL-92 Full. This must be the same as the value returned for the SQL_SQL_CONFORMANCE option in SQLGetInfo. In our case, I used “0” to indicate that the driver supports SQL-92 Entry level.

4. Research Issues

4.1. Overview

Generally, the research issue in this project is to investigate method for interacting with a Wireless Sensor Network using standard database tools. This will be done by building an ODBC interface for TinyDB and exploring its applicability for WSN interaction.

This project is not to build an alternative, but an extension to TinyDB. TinyDB is a popular sensor network application developed at Intel-Research Berkeley and UC Berkeley with the goal of providing a simple, SQL-like interface to extract data from the sensor network. It is a key component of a research effort exploring the relationships between database technologies and highly distributed, streaming, lossy, or real-world environments, such as networks of motes. Sensor networks have not previously been thought of as databases, so TinyDB applied an approach to represent the whole network as a database and distribute each data request to the individual nodes. In this case each node does not have to transmit any data until a request has been made. Furthermore, the node does not have to respond if the request is not of concern. TinyDB has an SQL-style processing language and supports in network aggregation. It uses a server system which optimizes the queries for both resource usage and reaction time, before it delivers them into the sensor network. Then, the queries execute on each node (called in-network aggregation) and the results are returned to the server.

Due to the popularity of TinyDB, lots of projects have been carried out to extend TinyDB in all sorts of aspects. Amol Deshpande and Joseph M. Hellerstein from UC Berkeley have done some work on Adaptive query processing schemes that attempt to reoptimize query plans during the course of query execution [Amo04]. The paper proposed a mechanism called STAIRs that allows the query engine to manipulate the state stored inside the operators and undo the effects of past routing decisions to achieve both high adaptivity and good performance.

Angelika Herbold and his team from UNSW have implemented another extension to TinyDB called ResTAG [Ang04], which provides fault-tolerant aggregate queries. The rough idea of this system is that if a sensor becomes mis-calibrated or physically compromised, the fault-tolerant queries use the network redundancy to reduce the confidence in values reported from that sensor. Queries implemented using ResTAG detect faulty nodes within a predictable threshold that depends on both the percentage and failure type of faulty nodes.

The current version of TinyDB provides a GUI that can understand SQL, but you can’t use that GUI to interface with a database. This project will open up a new world for people to use TinyDB by providing a standard ODBC interface which can work with all database tools available. Further more, a Wireless Sensor Network can be made to appear as a standard database table with each row representing a mote and each column representing a sensor or actuator. By sending a query to the table, you can easily get the current value of all sensor nodes just like doing a normal query on a database table.

4.2. Supporting TinyDB specific SQL

By changing the system to use an ODBC interface, the way of getting data from WSN also changes from continuous sampling to polling and response. This will introduce a problem of how to support TinyDB specific SQL extensions like SAMPLE PERIOD, STOP and TRIGGER ACTION.

The SAMPLE PERIOD SQL extension is used to specify the sampling rate of a continuous sampling query. With such an extension, it would be possible to specify the query “SELECT nodeid, light, temp FROM sensors SAMPLE PERIOD 1000”. This query specifies that each sensor should respond with its own identifier, the light intensity, and a temperature sample once per second. The sensors attribute in the query is the name of a virtual table containing one column for each attribute available and with all possible instants in time as rows. The attribute names have been stored in a catalog created at the same time as the sensor network. By calling that table a virtual table, what I mean is that there is no physically stored table in the node. The table is only temporary generated during the process of executing the query.

Since the SQL language has been extended with a continuous sampling query there must be a way to limit them. Therefore, another extension that has been introduced: the STOP statement. It is possible to use a query like “STOP QUERY id” where the id represents the identifier of the query the user wants to end. The identifier is a numerical value generated and unique to each query automatically and we can get the id of a query from its query result packet. Another way to limit a query is to use the FOR or the DURATION clauses described above. It is also possible to include a stopping condition as a triggering condition or an event.

4.2.1. Supporting the SAMPLE PERIOD statement

The ODBC interface introduces the problem of changing transaction paradigm to polling instead of continuous sampling which means the SAMPLE PERIOD statement is hard to support because multiple result sets can not be returned using the ODBC interface. However, we don’t want to lose the functionality of continuous sampling in our system. In order to provide a standard interface while not losing functionality, we need to make the ODBC driver capable of handling SAMPLE PERIOD statement and provide some way to store and retrieve the result sets created by a continuous sampling query.

Usually, there are two ways of handling DBMS specific SQL extensions and this design choice will affect how the driver processes the SQL statements. One way is that the driver must modify

SQL-92 SQL and the ODBC-defined escape sequences[6] to DBMS-specific SQL. But in real cases, because most SQL grammars are based on one or more of the various standards, most drivers do little or no work to meet this requirement. It often consists only of searching for the escape sequences defined by ODBC and replacing them with DBMS-specific grammar. The other way is to assume that the grammar is DBMS-specific when a driver encounters grammar it does not recognize and it therefore pass the SQL statement without modification to the data source for execution.

As the TinyDB specific SQL extensions are not in anyway similar to escape sequences, when the driver receives a query with TinyDB specific extensions, we can treat it as an unrecognized grammar which will be recognized and executed successfully by the TinyDB application.

Another aspect of the problem is that the SAMPLE PERIOD clause must exist in a TinyDB query to make the query successful, so that if the SQL statement is generated by some user applications like Microsoft EXCEL which simply do a “SELECT *” on the table you select, the driver needs to append the SQL statement with a SAMPLE PERIOD clause. The sample period can be specified by the user in a dialog box when modifying the DSN settings. The reason to specify the sample period in DSN setting is that user applications generally does not support the SAMPLE PERIOD clause so if a user tries to input the SAMPLE PERIOD directly in the user application, it is likely that he will get an error message. So I would expect no SAMPLE PERIOD in most SQL queries we get and the driver has to specify a default SAMPLE PERIOD. In this case, the driver can either displays a dialog box to prompt the user or retrieve this information from the registry. The former would require the user to input the sample period in every query they make which is not appropriate and displaying a dialog box is not always visible[7]. So, I put this default SAMPLE PERIOD setting in the DSN settings as shown in Figure 4.3, users only need to specify a default SAMPLE PERIOD once and every time the driver can retrieve it from the registry.

In this system, continuous sampling has been implemented by using a historical log table stored in a separate SQL Server database. Every time a query result comes back, it is published to the XMLBlaster Server and goes into two applications which are the ODBC driver and the TinyDB proxy. The ODBC driver only returns the first result set to the user application, and the TinyDB proxy stores the result sets in the historical log table. So, once we do a continuous sampling query, the first query result can be retrieved directly from the ODBC interface, and the rest of the result sets can be retrieved from the SQL Server.

4.2.2. Supporting the STOP statement

But as we know, once the sensor network receives a query, it will keep creating data until it

[6] A number of language features, such as outer joins and scalar function calls, are commonly implemented by DBMSs. However, the syntaxes for these features tend to be DBMS-specific, even when standard syntaxes are defined by the various standards bodies. Because of this, ODBC defines escape sequences that contain standard syntaxes for this kind of language features.
[7] In certain applications like SQL Server 2000, if the driver displays a dialog box, the dialog box will not be visible. This is possibly because SQL Server runs as a system service and does not have a window.

receives a STOP command, so if the query only wants a single sample, the driver needs to send a STOP command to the sensor network with the correct query ID once the query result comes back.

During the development, a problem has been noticed that the STOP command sometimes doesn’t work which means the STOP has to be sent several times to reliably make the sensor network actually stop. Through investigation, this problem is due to the fact that TinyDB uses a hybrid approach to disseminate and to stop the queries [Ibr05]. When a new query is sent to the sensor network, TinyDB first floods new queries into the network; this reaches most nodes very quickly. Then the system uses epidemic approaches to reach the remaining nodes which are usually far away from the base station and can not receive the query directly. In the epidemic phase, sensors send query result ids to their neighbors. If a sensor gets a query id that it did not receive, it assumes that it missed the query message due to a poor signal and sends a query request message to the node that has the query message. In response, the node sends the actual query message to the requesting node. This will make sure that all nodes in the sensor network get the query message finally, either directly from the base station or indirectly from its neighbors.

The epidemic algorithm is intended to make the sensor network reliable, but it introduce a new problem. Sometimes one or more nodes will not receive the STOP command due to a poor signal. They will keep sending query result ids to their neighbors and their neighbors will perceive that it is a new query and start their job again. In other words, if one node has not been stopped, it will propagate the old query to the whole sensor network and restart the query.

4.2.3. Supporting the TRIGGER ACTION statement

There is a special SQL extension in TinyDB called TRIGGER ACTION that allows some command to be executed when a result is produced. Imagine an application where the user wants to sound an alarm whenever the temperature near a sensor goes above some threshold. This can be accomplished by adding a TRIGGER ACTION:

```
SELECT temp
FROM sensors
WHERE temp > thresh
TRIGGER ACTION SetSnd(512)
```

In the ODBC driver, the TRIGGER ACTION statement can be supported in two ways, the first way is when the user application sends a TRIGGER ACTION statement to the ODBC driver, and it forwards the statement to TinyDB proxy and relies on TinyDB proxy to translate the statement into mote commands. This requires the user application and the TinyDB proxy to support the TRIGGER ACTION statement. Another way is to translate the UPDATE statement into a TRIGGER ACTION statement which means the user application can send an ordinary UPDATE query to TRIGGER an ACTION on the sensors. For example, a user application can sound an alarm by sending a separate query like:

```
UPDATE sensors
SET sound=512
WHERE temp > thresh
```

Because TinyDB does not support the UPDATE statement, we can safely translate the UPDATE queries into TRIGGER ACTION. This allows us to use UPDATE as an alternative to TRIGGER ACTION and the user applications can do the same thing without having to support the TRIGGER ACTION statement. Furthermore, UPDATE statement can be used as an analog output, for example, we can switch things on and off by updating a value in the table.

4.3. Mapping WSN to the database paradigm

One of the most important research issues in this project is how effectively can a sensor network be mapped to database paradigm given the fundamental differences. In the chapters below, I'll discuss how I mapped the sensor network to each step in the paradigm of getting data from a normally available database.

4.3.1. Connecting to the database

The first step in any application is to connect to the data source. This phase, including the functions it requires, is shown in the following illustration.

Step 1: Connect
SQLAllocHandle(ENV)
SQLSetEnvAttr
SQLAllocHandle(DBC)
SQLDriverConnect
SQLSetConnectAttr
SQLGetInfo
SQLGetFunctions
SQLAllocHandle(STMT)
SQLSetStmtAttr

Figure 4.1 ODBC function involved in connecting to database

The first step in connecting to the data source is to load the Driver Manager and allocate the environment handle with SQLAllocHandle. The environment structure generally contains global variables in the driver and a list of all the connections to the driver. So when the user application calls this function, the driver just creates an instance of the environment class and returns a pointer to it.

The application then registers the version of ODBC to which it conforms by calling SQLSetEnvAttr with the SQL_ATTR_APP_ODBC_VER environment attribute. The driver is developed agains ODBC version 3 which is the latest ODBC verison, so it returns SQL_SUCCESS when the application also conforms to ODBC version3.

Next, the application allocates a connection handle with SQLAllocHandle and connects to the data source with SQLConnect, SQLDriverConnect, or SQLBrowseConnect. When these functions are called, the driver is supposed to connect to the data source which is TinyDB, but sooner I found out that I should not connect to TinyDB proxy or XMLBlaster at this stage because the query statement is not created at this stage and creating a socket in this stage and writing to the socket in another stage is more complicated and slower than making the connection when executing the query statement.

The application then sets any connection attributes, such as whether to manually commit transactions. This function returns without taking any action in the ODBC driver for WSN because these attributes can not be mapped onto a sensor network.

After that, the application use SQLGetInfo and SQLGetFunctions to discover the capabilities of the driver. In SQLGetFunctions, the driver returns an array that is treated by the Driver Manager as a 4,000-bit bitmap that can be used to determine whether an ODBC 3.*x* or earlier function is supported. In SQLGetInfo, things are more complicated, the application uses this function to ask all sorts of questions like the driver’s name, which version it is and what behavior does the driver expect from the application. Among these questions, I think the most important one is SQL_DATA_SOURCE_READ_ONLY which asks if the data source is read only. If the driver answers NO to this question, the application will not try to UPDATE or INSERT to the table. This is desirable if we only want to use the driver to retrieve data from the sensor network. But if we want to output data to the sensor network, we should answer YES to the question.

The last step is for the application to allocate a statement handle with SQLAllocHandle, and set statement attributes, such as the cursor type, with SQLSetStmtAttr. When these functions are called, the driver allocates memory for a statement structure which contains several things including a connection handle to the connection it uses and some error number and error message later used to answer diagnostic calls.

4.3.2. Building and Executing an SQL statement

The major step in the ODBC process is to build and execute an SQL statement, as shown in the following illustration. The methods used to perform this step are likely to vary tremendously. The application might prompt the user to enter an SQL statement, build an SQL statement based on user input, or use a hard-coded SQL statement. But the most usually way is to first use SQLTables to get the list of available tables and prompt user to specify which table to select and what attributes to select on that table.

Step 2: Building and Executing an SQL statement
SQLTables
SQLNumResultCols
SQLDescribeCol
SQLBindCol
SQLFetch
SQLExecuteDirect

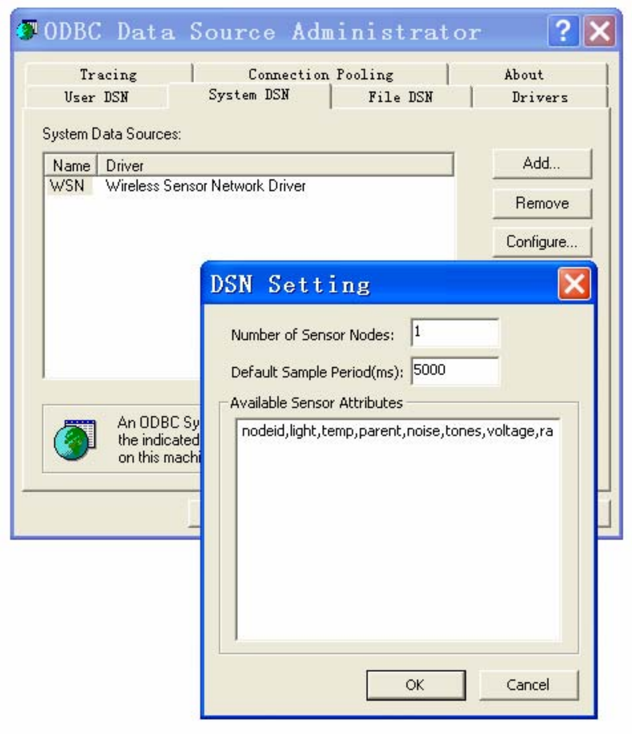
Figure 4.2 ODBC function involved in building and executing SQL statements

The SQLTables function returns SQL_SUCCESS when it successfully creates a query result which contains the list of tables and their attributes. The application then has to fetch this information like an ordinary query result. For more details about fetching a query result, please refer to section 4.3.3. The result set contains the following columns:

Column name	**Column number**	**Data type**	**Comments**
TABLE_CAT	1	Varchar	Catalog name; I simply return "TinyDB" as the catalog name.
TABLE_SCHEM	2	Varchar	Schema name; I returned NULL because it is not applicable.
TABLE_NAME	3	Varchar	Table name. The table have to be the same name as the Ultralite device so that the application will generate a query to select attributes from it.
TABLE_TYPE	4	Varchar	Table type name; one of the following: "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM", or a data source–specific type name. I simply return "TABLE".
REMARKS	5	Varchar	A description of the table. Can be an empty string.

After the SQL statement is built, the statement is executed with SQLExecDirect. SQLExecuteDirect gets the SQL statement from the application and replaces the "*" with all available attributes so that only those available in the sensor network are selected. If the SQL statement tries to select attributes which doesn't exist in the sensor network, an error is returned.

Because the driver does not know what attributes are available in the sensor network. I have created a dialog box in the DSN setting as shown in Figure 4.3. Users should be able to change this information according to what sensor boards the have on the sensor nodes.



System Data Sources:

Name	Driver
WSN	Wireless Sensor Network Driver

Number of Sensor Nodes: 1

Default Sample Period(ms): 5000

Available Sensor Attributes

nodeid,light,temp,parent,noise,tones,voltage,ra

Figure 4.3. DSN Settings Dialog Box

The ODBC driver then appends a “SAMPLE PERIOD” clause to the statement so that TinyDB proxy will accept it. The default sample period is also read from the DSN settings as shown in Figure 4.3. For more details, please refer to the previous chapter “Supporting TinyDB specific SQL”. The completed SQL statement will look like this:

```
SELECT nodeid, temp FROM Jun SAMPLE PERIOD 5000
```

After the SQL statement is built, SQLExecuteDirect creates a socket to the TinyDB proxy server and send the SQL statement to it. Wait for the return code to come back and map the error code to an ODBC error state.

If the SQL statement is sent successfully, SQLExecuteDirect then creates a connection to XMLBlaster server and subscribes to the topic “mobile” which the backend server uses to publish messages from the sensor network. Once the subscription is successful, the query result will come back asynchronously. This usually takes about 20 seconds because the sensor network takes a long time to start all the sensors and to do the measurements. Once the result comes back to the subscriber which is our ODBC driver, a query result is created and SQLExecuteDirect can finally return SQL_SUCCESS.

4.3.3. Fetching a query result

The application's next action is to fetch the query result.

Step 3: Fetch result
SQLNumResultCols
SQLDescribeCol
SQLBindCol
SQLFetch
SQLGetData

Figure 4.4 ODBC function involved in fetching results

In this step, the application first calls SQLNumResultCols to determine the number of columns in the result set. Not all applications call this function as it is frequently possible to infer this number from the SQL directly.

Next, the application retrieves the name, data type, precision, and scale of each result set column with SQLDescribeCol. Again, this is not necessary for applications that already know this information. The application passes this information to SQLBindCol, which binds an application variable to a column in the result set.

The application now calls SQLFetch to retrieve the first row of data and place the data from that row in the variables bound with SQLBindCol. If data in the row is not returned, it then calls SQLGetData to retrieve more data for that row. The application continues to call SQLFetch and SQLGetData to retrieve additional rows of data until SQLFetch returns SQL_NO_DATA.

The number of result rows is the same as the number of sensor nodes. Because the driver does not know how many sensor nodes there are in the sensor network, I have chosen to have this information specified by the user in the DSN setting as shown in Figure 4.3. The driver will fetch exactly one result message for each sensor node from the sensor network by checking the nodeid in the query result message.

4.4. Power Consumption

Power consumption is one of the most important issues in Wireless Sensor Network, since it determines battery use, and therefore the lifetime of the network. The major source of energy consumption in the sensor node is the wireless communication, so if the sensor node can be completely shut down for a period of time, considerable energy can be saved. As noted in [Buo05], TinyDB initially left nodes on all the time, which limited lifetime to just a few days. To meet TASK network lifetime requirements, they have explored two power management solutions for TinyDB: duty cycling and low-power listening. In duty cycling, nodes are awake for a short, synchronized period each sample interval. This period must be long enough for a node to sample sensors, send its data and forward data for its children. Ideally, this period would be adaptive, but currently TinyDB uses a fixed period which is set by adding an SAMPLE PERIOD extension to SQL. In low-power listening, each node wakes up periodically to sample the channel for traffic and goes right back to sleep if there is nothing being sent. In this mode, nodes send messages with a preamble that is at least as long as the period with which listeners sample the channel to ensure that the messages will be received by the destination nodes. The advantage of low-power listening is that the TinyDB application layer no longer needs to manage the scheduling of wake and sleep. The disadvantage is that it substantially increases transmission cost. For more details including performance comparison of these two approaches, please refer to TASK: Sensor Network in a Box [Buo05] Section IV-C.

Generally, there are three factors that contribute to the lifetime of motes. First, nodes have a higher base power consumption when idle: according to [Buo05] the low power listening mode consumes 1mW when “idle” (checking every 100ms for a message), while the duty cycling mode consumes 0.45mW when “off” (waiting for the next epoch). Second, sending messages costs more because they must have a 100ms preamble. Third, nodes will wake up from low-power mode to listen to messages which were not intended for them, this will help by spending a higher proportion of each epoch awake.

In the current ETDEMO system [Etd05], temperature information will keep going into database even when we don’t need it because the system needs to provide real-time information but it doesn’t know when the user will need this information. With an ODBC interface, we get a new way to query data which is polling and response, so that the sensor network can be made to do the measurement only once when demanded. In other words, we wake up the sensor network only when we need to know if a room is occupied and the sensor network is put to sleep immediately once the transaction is complete. This will potentially help to save some energy if we can find a good schedule scheme to control each sensor node to wake up when a query is received in low-power listening mode. However, there will be a cost in the increased latency as discussed in section 4.5.

4.5. Latency

Latency is another important research issue in this project. The latency here is the time taken form a SQL query being sent out by user, to the time data come back from WSN and is returned by the ODBC interface. As we change the way of getting data from polling instead of continuous sampling, the latency issue has become more important than before because formerly we could get historical data from local database directly with a very low latency but now the data has to be acquired from the sensor network before it can be returned by the ODBC interface. In other words, the old way is like buying a computer from an ordinary shop, where lots of computers are in the stock. So you get your computer as soon as you pay. However getting data from the ODBC driver is like buying a computer from DELL. DELL does not have any computers in their stock, instead they produce the computer after you pay for it. Buying a computer from DELL of course takes more time, but it is also a good way of buying computers because it gives you the best price.

There are a number of factors contributing to the overall latency. These include:

- The latency introduced by the ODBC driver and DBMS.
- The latency introduced by the TinyDB Proxy.

- The latency introduced by the Sensor Network.
- The latency introduced by the Mobile Station.
- The latency introduced by radio resource procedures

The overall latency in this system is around 50 seconds, which can be divided into four parts as shown in Figure 4.5.

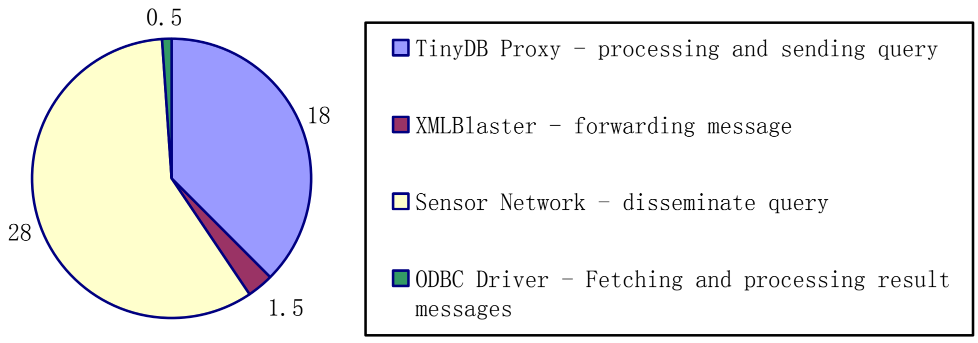


Figure 4.5 Sources of latency (in seconds)

The latency introduced by WSN is generally because of the power management which would shut down the motes for a period of time during which it won't do anything. The longer a mote sleeps the less power it will consume, but the more latency will occur because we need to wait for the mote to wake up. So, it is a tradeoff between latency and energy efficiency. Also, because the processing power is limited on the motes, it takes a few seconds to process the query and measure sensor attributes. And the processing time is depend on the number of attributes in the query message and the number of motes we have. When a query selects many attributes, it would take a long time for the sensor network to process it because TinyDB proxy will generate a query packet for each of the attributes. According to my test, it takes about 28 seconds for the sensor network to start all the sensor nodes and send back the first query result.

The latency introduced by TinyDB proxy is also large. For example, in my experiment I selected 12 attributes from the sensor network by a query like this:

> SELECT nodeid, light, temp, parent, noise, tones, voltage, rawtone, rawmic, freeram, depth, qual FROM WSN SAMPLE PERIOD 5000

The TinyDB proxy would take about 22 seconds to process and send the query and it sometimes returns Timeout error. This large latency is because TinyDB proxy is written in java which is slow and the message has to go through GPRS network which has low band width and large latency. Another reason for the large latency is that TinyDB proxy generates one query packet for each of the attributes in a query and only one query packet is sent at a time which means it doesn't send another query packet until it receives the response for the last one from the sensor network. This means we have at least 12 times the latency of GPRS network if we select 12 attributes from the sensor network.

The time taken to transmit data over the GPRS network is probably the major source of latency which is over 500 ms for most users, and can be as long as 4 seconds [Dav03] for some while DSL only has a latency of 20 ms or so. If we use the new CDMA device called CDM820 [Cdcs05] instead of Ultralite [Cdcs03], the latency could be a little smaller as CDMA 1x will deliver a latency of 100-300ms which is roughly the same as an analog modem connection. David Adam [Dav03] has found CDMA2000 1x to be about 3-5 times faster (bandwidth) than dial-up, with comparable latency.

The latency introduced by the ODBC driver and the DBMS is the time taken by the DBMS to forward the query to the ODBC driver through the driver manager, and the time taken waiting for the query result and translating into a database table. Because the driver is written in C++, this latency is generally less than one second.

Mobile Station delay is the time taken by the Mobile Station to process an IP datagram and request radio resource. This includes the delay from the back-end server to the Mobile Station, and the Mobile Station processing time. This delay is typically less than approximately 100ms which may vary depends on the Mobile Station, and the service provider.

Radio resource procedures are the major source of delay in GPRS. In order for the Mobile Station to be capable of sending or receiving data, radio resource known as a Temporary Block Flow (TBF) must be made available to the user. If a TBF is currently active then the Mobile Station may use it hence minimizing the delay. However, if no TBF is established then the Mobile Station and network must exchange signaling messages in an attempt to establish a TBF. The time taken to successfully achieve an active TBF will depend on the availability of radio resources and will be different for the uplink and downlink directions. Once established, the TBF will generally remain active for as long as data is made available to the layer (i.e. for as long as there are LLC frames to transmit).

Effective data throughput (over-the-air delay) is the rate at which user data is physically transmitted between the Mobile Station and the handset over an active TBF. The delay associated with this throughput is directly related to the size of the IP datagram being sent. Smaller packets cause less delay. The delay is proportionally reduced when multiple timeslots are used. The effective throughput is also dependent on the number of re-transmissions due to poor signal strength. The time taken to re-transmit erroneously received information will affect the size of the latency.

5. Conclusion

After examination of the field, it was determined that a standard ODBC interface to TinyDB Remote Wireless Sensor Networks would be advantageous to many applications. An ODBC driver was created to support a subset of the standard SQL-92 grammar as well as the TinyDB specific SQL extensions. With such a driver, we are able to make the WSN appear as a standard database table, so that most normally available database tools can get access to the sensor data without any modification. Also, it is possible to interact with a wireless sensor network using standard database tools but without requiring a database to be present.

In this project, I faced many technical difficulties. One of the most difficult issues is to understand how the ODBC interface works with the user applications and the underlying DBMS. The ODBC interface is designed to work with all DBMSes which means it has abstracted all functionalities that a normal database can provide. Actually, the interface provides more than enough functionality that no DBMS can implement them all. So generally, the interface is very complicated and takes a long time to understand how it works.

Another technical difficulty is the compatibility with various user applications. The ODBC interface consists of 76 ODBC functions. Different user applications may call these functions in a slightly different way which makes the development hard. The driver usually works with one application but doesn't necessarily work with another. Creating a driver that works with all user applications is a challenge.

Memory corruption is also a difficult technical issue which may cause problems like unexpected result or even make the user application crash. Since the driver resides in the same memory space of the user application, nothing will stop it from writing into the memory space of the user application. These bugs are usually the most difficult because you never know which part of the code makes the user application crash since there's no way to debug a user application. Also, these bugs make trouble in release version but they usually disappear in debug version which also makes it hard to find and fix them.

The implemented ODBC driver is not a finished product, however, it can at this stage work with various normally available database tools including Microsoft Excel. TinyDB specific SQL extensions like SAMPLE PERIOD are supported in the driver. Continuous sampling is also supported by returning data to a separate historical log stored in SQL Server. Instead of return the query results through ODBC interface, I choose to store the continuously sampled data in a separate databse because the user application unloads the ODBC driver after a query result is returned. Therefore multiple results can not be returned in the ODBC driver.

Some supporting technologies are used in this project, including the XMLBlaster which is a message oriented middleware, the TinyDB proxy which is a java application that translate SQL query into TinyDB message, and the back-end server which is a message routing system. These supporting technologies simplified the driver development and provided more functionality to the whole system.

This report also discussed some benefits and problems that might be of concern due to the new way of getting data from WSN. Usability is a major benefit because the standard ODBC interface is widely supported and familiar to ordinary users. Power consumption is also a benefit after utilization of the new ODBC driver because the change from continuous sampling to polling data will require that the motes acquire data mush less often for many applications. On the other hand, user have to wait for a long time as the data is collected on the field after the query is issued, thus introducing a larger latency.

6. Future Works

To make the ODBC driver reaches the next maturity level, I’m investigating to improve its compatibility with other database tools. This is very hard because there are lots of user applications designed for ODBC and each one may have different compatibility problems.

Further more, we need to make the driver support other TinyDB specific SQL extensions like EPOCH DURATION and FOR, we have also discussed using UPDATE statement as an alternative to TRIGGER ACITON and the possibility to use UPDATE statement to do analog output on the wireless sensors.

Another thing we can do in the future is to find a good way to get the device name of the Ultralite. As the Ultralite name is actually used as the table name in the SQL statement, the driver can either get it from the back-end server or have the user input it in the DSN setting. The former way would require the driver talks to the back-end server and get a list of online devices and it would be a problem to tell which one is the Ultralite that connected to a sensor network as we have some Ultralites running for other projects. The latter way is relatively simple, but it requires the user to know which Ultralite is connected to the sensor network and the device name of that Ultralite which is not obvious because the device name can only be seen on the back-end server.

Appendix A: SQL Minimum Grammar

This section describes the minimum SQL syntax that an ODBC driver must support. The syntax described in this section is a subset of the Entry level syntax of SQL-92.

An application can use any of the syntax in this section and be assured that any ODBC-compliant driver will support that syntax. To determine whether additional features of SQL-92 not in this section are supported, the application should call SQLGetInfo with the SQL_SQL_CONFORMANCE information type. Even if the driver does not conform to any SQL-92 conformance level, an application can still use the syntax described in this section. If a driver conforms to an SQL-92 level, on the other hand, it supports all syntax included in that level. This includes the syntax in this section because the minimum grammar described here is a pure subset of the lowest SQL-92 conformance level. Once the application knows the SQL-92 level supported, it can determine whether a higher-level feature is supported (if any) by calling SQLGetInfo with the individual information type corresponding to that feature.

Drivers that work only with read-only data sources might not support those parts of the grammar included in this section that deal with changing data. An application can determine if a data source is read-only by calling SQLGetInfo with the SQL_DATA_SOURCE_READ_ONLY information type.

Statements

create-table-statement ::=

CREATE TABLE *base-table-name*

(*column-identifier data-type* [,*column-identifier data-type*]...)

Important As a *data-type* in a *create-table-statement*, applications must use a data type from the TYPE_NAME column of the result set returned by **SQLGetTypeInfo**.

delete-statement-searched ::=

DELETE FROM *table-name* [WHERE *search-condition*]

drop-table-statement ::=

DROP TABLE *base-table-name*

insert-statement ::=

INSERT INTO *table-name* [(*column-identifier* [, *column-identifier*]...)]

VALUES (*insert-value*[, *insert-value*]...)

select-statement ::=

SELECT [ALL | DISTINCT] *select-list*

FROM *table-reference-list*

```
    [WHERE search-condition]
    [order-by-clause]

statement ::= create-table-statement
    | delete-statement-searched
    | drop-table-statement
    | insert-statement
    | select-statement
    | update-statement-searched

update-statement-searched
    UPDATE table-name
    SET column-identifier = {expression | NULL }
        [, column-identifier = {expression | NULL}]...
    [WHERE search-condition]
```

Appendix B: Core Level Conformance

All ODBC drivers must exhibit at least Core-level interface conformance. Because the features in the Core level are those required by most generic interoperable applications, the driver can work with such applications. The features in the Core level also correspond to the features defined in the ISO CLI specification and to the nonoptional features defined in the X/Open CLI specification. A Core-level interface‑conformant ODBC driver allows the application to do all of the following:

- Allocate and free all types of handles, by calling **SQLAllocHandle** and **SQLFreeHandle**.
- Use all forms of the **SQLFreeStmt** function.
- Bind result set columns, by calling **SQLBindCol**.
- Handle dynamic parameters, including arrays of parameters, in the input direction only, by calling **SQLBindParameter** and **SQLNumParams**.
- Specify a bind offset.
- Use the data-at-execution dialog, involving calls to **SQLParamData** and **SQLPutData**.
- Manage cursors and cursor names, by calling **SQLCloseCursor**, **SQLGetCursorName**, and **SQLSetCursorName**.
- Gain access to the description (metadata) of result sets, by calling **SQLColAttribute**, **SQLDescribeCol**, **SQLNumResultCols**, and **SQLRowCount**.
- Query the data dictionary, by calling the catalog functions **SQLColumns**, **SQLGetTypeInfo**, **SQLStatistics**, and **SQLTables**.

  The driver is not required to support multipart names of database tables and views. However, certain features of the SQL-92 specification, such as column qualification and names of indexes, are syntactically comparable to multipart naming. The present list of ODBC features is not intended to introduce new options into these aspects of SQL-92.

- Manage data sources and connections, by calling **SQLConnect**, **SQLDataSources**, **SQLDisconnect**, and **SQLDriverConnect**. Obtain information on drivers, no matter which ODBC level they support, by calling **SQLDrivers**.
- Prepare and execute SQL statements, by calling **SQLExecDirect**, **SQLExecute**, and **SQLPrepare**.
- Fetch one row of a result set or multiple rows, in the forward direction only, by calling **SQLFetch** or by calling **SQLFetchScroll** with the *FetchOrientation* argument set to SQL_FETCH_NEXT.
- Obtain an unbound column in parts, by calling **SQLGetData**.
- Obtain current values of all attributes, by calling **SQLGetConnectAttr**, **SQLGetEnvAttr**, and **SQLGetStmtAttr**, and set all attributes to their default values and

set certain attributes to nondefault values by calling **SQLSetConnectAttr**, **SQLSetEnvAttr**, and **SQLSetStmtAttr**.

- Manipulate certain fields of descriptors, by calling **SQLCopyDesc**, **SQLGetDescField**, **SQLGetDescRec**, **SQLSetDescField**, and **SQLSetDescRec**.
- Obtain diagnostic information, by calling **SQLGetDiagField** and **SQLGetDiagRec**.
- Detect driver capabilities, by calling **SQLGetFunctions** and **SQLGetInfo**. Also, detect the result of any text substitutions made to an SQL statement before it is sent to the data source, by calling **SQLNativeSql**.
- Use the syntax of **SQLEndTran** to commit a transaction. A Core-level driver need not support true transactions; therefore, the application cannot specify SQL_ROLLBACK nor SQL_AUTOCOMMIT_OFF for the SQL_ATTR_AUTOCOMMIT connection attribute.
- Call **SQLCancel** to cancel the data-at-execution dialog and, in multithread environments, to cancel an ODBC function executing in another thread. Core-level interface conformance does not mandate support for asynchronous execution of functions, nor the use of **SQLCancel** to cancel an ODBC function executing asynchronously. Neither the platform nor the ODBC driver need be multithread for the driver to conduct independent activities at the same time. However, in multithread environments, the ODBC driver must be thread-safe. Serialization of requests from the application is a conformant way to implement this specification, even though it might create serious performance problems.
- Obtain the SQL_BEST_ROWID row-identifying column of tables, by calling **SQLSpecialColumns**.

Important ODBC Drivers must implement the functions in the Core interface conformance level.

REFERENCES

[Amo04] Amol Deshpande, Joseph M. Hellerstein. “Lifting the Burden of History from Adaptive Query Processing”. VLDB 2004.

[Ang04] Angelika Herbold1, Thierry Lamarre , Nirupama Bulusu and Sanjay Jha, “Resilient Event Detection in Wireless Sensor Networks”, http://www.cse.unsw.edu.au/~sensar/publications/niru_issnip04.pdf

[Asa03] Asada, H. H., Shaltis, P., Reisner, A., Rhee, S., und Hutchinson, R. C. “Mobile Monitoring with Wearable Photoplethymographic Biosensors”. IEEE EMB Magazine. 22(3):28–. 2003.

[Bak03] Doug Palmer, “SCADA Backend Architecture Documentation”, June 2003 http://mobile.act.cmis.csiro.au/Backend/architecture.doc

[Buo05] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, S.Madden, “TASK: Sensor Network in a Box” , IRB-TR-04-021, April 2005.

[Cdcs03] Call Direct Cellular Solutions Pty Ltd. “Ultralite E it GSM/GPRS Intelligent Terminal”, http://www.call-direct.com.au/files/UltraliteEit.pdf

[Cdcs05] Call Direct Cellular Solutions Pty Ltd. “CDMA 1XRTT Cellular Modem Model CDM820”, http://www.call-direct.com.au/files/CDM-820_cutsheet.pdf

[Co04] Rachel Cardell-Oliver. “Why flooding is unreliable in multi-hop, wireless networks”, February 2004. http://www.cs.uwa.edu.au/research/Technical_Reports/UWA-CSSE-04-001/Report-04-001.pdf,

[Etd05] CSIRO Energy Center, Distributed Energy Management and Control Project, http://etdemo.hopto.org/

[Fen01] W. Feng, J. Wadpole, W Feng, and C. Pu, “Moving Towards Massively Scalable Video-Based Sensor Networks”, Large Scale Networking Workshop, 2001.

[Ibr05] Dr. Ibrahim Körpeoglu, Prof. Özgür Ulusoy, “Improving Reliability of Data Dissemination in Sensor Networks Through Epidemic Algorithms”, Bilkent University, 2005.

[Mar04] K. Martinez, J.K. Hart, and R. Ong. “Environmental Sensor Networks”. Computer, August 2004.

[Ms05] Microsoft MSDN Library Data Access. http://www.microsoft.com/data/odbc

[Msd05] ODBC Application Steps chart. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/odbcbasic_application_steps.asp

[Msdn05] MSDN ODBC Programmer's Reference http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/odbcabout_this_manual.asp

[Nem05] National Electricity Managerment Company, http://www.nemmco.com.au/

[Dav03] David Adams (david@osnews.com), “The 3G/2.5G Controversy: GSM vs. CDMA”, 2003. http://www.nmcx.com/story.php?news_id=3091

[Pol03] Polastre,J. “Design and Implementation of Wireless Sensor Networks for Habitat Monitoring”, UC Berkeley. 2003.

[Ram03] Parmesh Ramanathan. “Communication support for location-centric collaborative signal processing in sensor networks”, April 2003. http://www.ece.wisc.edu/~sensit/publications/dimacs01.ppt

[Ref05] ODBC Programmer’s Reference, Multithreading, MSDN 2005. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/odbcmultithreading.asp

[Wik05] Wikipedia. Supply and demand. http://en.wikipedia.org/wiki/Supply_and_demand, April 2005.