Data Monitoring Web Services in a Virtual Lab Environment

By

Ashish Sadashiv Kulkarni

B.Tech. Chemical Engineering, IIT Madras, 1999
M.S. Chemical Engineering, PennState University, 2001

Submitted to the Department of Civil and Environmental Engineering in partial fulfillment of the requirements for the Degree of

MASTER OF ENGINEERING IN CIVIL AND ENVIRONMENTAL ENGINEERING

AT THE

MASSACHUSSETS INSTITUTE OF TECHNOLOGY

JUNE 2002

Author.....

Ashish Sadashiv Kulkarni
Department of Civil and Environmental Engineering
May 13, 2001

Certified.....

Kevin Amaratunga
Assistant Professor, Department of Civil and Environmental Engineering

Accepted by.....

Oral Buyukozturk
Chairman, Departmental Committee in Graduate Studies

Data Monitoring Web Services in a Virtual Lab Environment

By

Ashish Sadashiv Kulkarni

Submitted to the Department of Civil and Environmental Engineering on May 24, 2002 in partial fulfillment of the requirements for the Degree of

Master of Engineering in Civil and Environmental Engineering

ABSTRACT

Environmental issues have become of prime concern due to dramatic increase in the pollution levels in all parts of the world. Underlying aquifer flow in environmentally sensitive places plays an important role in characterizing the environmental condition of the place. There is thus a pressing need for monitoring and real time analysis of hydrological data over areas of environmental interest. Coupling the emerging sensing and wireless technologies with an internet infrastructure can enable efficient data monitoring and real-time analysis of environmental conditions over an area of interest. Information gathered from various data sources regarding the change in water level and quality during various seasons can then be used to characterize trends in the physical, chemical and biological condition of the environment.

Efficient real-time monitoring furnished with fast data rendering and decision-making capabilities can go a long way in monitoring Civil and Environmental Engineering infrastructure. The speed and reliability necessary for such a task can be achieved only by using a distributed infrastructure, with dedicated resources to data acquisition, archival and rendering. Distributed development technologies like DCOM, CORBA, RMI and SOAP, essentially extensions of simple RPC protocols, provide the interconnectivity between different components of such a distributed infrastructure. The present work discusses these distributed development technologies and compares them in the context of the Smartwells project.

Thesis Supervisor: Dr. Kevin Amaratunga

Title: Assistant Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my family who provided the inspiration and encouragement for me to come to M.I.T.

I would like to thank my thesis advisor Dr. Kevin Amaratunga for his tutelage and assistance throughout this learning process. I would also like to thank Dr. Eric Adams, my advisor for the Smartwells project for his help and guidance during the project. Also, I would like to express special gratitude to fellow M.S. student Ragunathan Sudarshan, for his help and support throughout the course of the project.

I am also very thankful to Raghu Narayan, fellow partner in the Smartwells Project for the constant support during my year at MIT.

Finally, I would like to thank God for blessing me with the opportunity to pursue my education this far.

Table of Contents

Table of contents..... 7

List of Figures..... 10

List of Tables 11

Chapter 1 Introduction..... 12

1.1 Motivation..... 12

1.2 Purpose 13

1.3 Layout of the Thesis 14

Chapter 2 Smartwells Project 15

2.1 Project Overview 15

2.1.1 Laboratory Prototype 16

2.1.2 Deployment outside Parsons Lab 17

2.1.3 Planned Field Deployment at Waquoit Bay Reserve 18

2.2 Sensors 19

2.2.1 Water Level Sensors 19

2.2.2 Conductivity Sensors 22

2.2.3 Tipping Bucket Rain Gauge 24

2.3 Field Point Data Acquisition System 27

2.3.1 Input Output Module 28

2.3.2 Network Module 29

2.4 Data Collection 31

2.4.1 Wireless Architecture 32

2.5 Software for Data Acquisition 33

2.5.1 LabWindows/CVI Interface 35

2.5.2 DataSockets API 38

2.5.3 Archiving Data 40

2.5.4 Data Visualization 43

2.6 Need for Distributed Architecture 48

Chapter 3 Distributed Development – Technology Overview 50
3.1 CORBA 51
3.2 DCOM 51
3.3 JAVA/RMI 52
3.4 Middleware 53
3.5 Application Sample - Smartwells Data Archive Server and Client 53
3.6 Implementing the IDL Interface 54
3.7 Fundamentals of Remoting 56
3.8 Implementing the Distributed Object Client 56
3.9 Implementing the Distributed Object Server 59
3.10 The Server Main Programs 60
3.11 Newer Technologies and their Comparison 66
3.12 Introduction to SOAP 67
3.13 Conclusion 68

Chapter 4 SOAP and Web Services Architecture 70
4.1 Overview 70
4.1.1 Evolution of Web Services 70
4.1.2 Network Tiers 71
4.1.3 XML: The key to describing web services 71
4.1.4 Loosely Coupled Systems 72
4.1.5 Web Services and CORBA 73
4.1.6 Publish, Bind and Find Model 73
4.2 Building Web Services with SOAP 75
4.2.1 SOAP Clients and Servers 75
4.2.2 SOAP and Java Technology 75
4.2.3 A SOAP Use Case 75
4.3 Role of SOAP in Web Services Architecure 77
4.3.1 Message Format 78

4.3.2 Anatomy of a SOAP Envelope 79
4.3.3 Namespaces 80
4.3.4 Header 80
4.3.5 Body 81
4.3.6 SOAP-RPC 81
4.4 Summary 83

Chapter 5 Field Implementation and Conclusions 84
5.1 Field Implementation at Waquoit Bay 84
5.1.1 Proposed Plan of Implementation..... 84
5.2 Conclusion 86

References..... 88

Appendix A 89

List of Figures

2.1	Smartwells Laboratory Set-Up.....	17
2.2	Parsons Lab.....	18
2.3	Waquoit Bay Reserve, Cape Cod.....	19
2.4	Water-level Sensor.....	20
2.5	Conductivity Sensor.....	23
2.6	Rain Gauge.....	25
2.7	Complete Fieldpoint Data Acquisition System.....	27
2.8	FP-AI_110 I/O module.....	28
2.9	FP-1600 Network Module –NI.....	30
2.10	Wireless System Architecture	33
2.11	FieldPoint Explorer Architectire.....	35
2.12	Interface using CVI	37
2.13	DataSocket Model	39
2.14	Table Layout for MS-SQL Server 2000	41
2.15	Data Model for MS-SQL Server 2000	41
2.16	Water Monitoring Applet	44
2.17	Well Properties Gradient Applet	45
2.18	Water Level and Conductivity Table Web Service	46
2.19	Water Level, Conductivity and Precipitation Applets	47
2.20	RMI Model	49
3.1	Difference between DCOM and SOAP Architecture	68
4.1	SOAP Use-Case Diagram	76
5.1	Waquoit Bay Sensor Deployment	85
5.2	Waquoit Bay Server Deployment	86

List of Tables

2.1 Specifications for water level sensor WL300..... 22

2.2 Specifications for conductivity level sensor 24

2.3 Specifications for Rain Gauge RG600..... 26

2.4 Specifications for FP-AI-110 I/O Module 28

2.5 Sampling Rates for FP-AI-110..... 29

2.6 Specifications for FP-1600 Network Module 30

2.7 Transfer Rates for FP-1000 [with FP-AI-110 I/O module]..... 31

2.8 Services running on the Smartwells server 35

3.1 Comparison of DCOM, CORBA, RMI and SOAP..... 68

CHAPTER 1

INTRODUCTION

The purpose of this thesis is to examine the use of different distributed development technologies and their application in the context of a real-time data monitoring project. The project in consideration is 'Smartwells' -a student initiative in the Department of Civil and Environmental Engineering at the Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts. The 'Smartwells' project, sponsored by the MIT-Microsoft *i-Campus* Alliance, started in May 2001 and presently consists of three Master's students and two faculty advisors, Prof. Kevin Amaratunga and Dr. Eric Adams.

The 'Smartwells' Project introduces the virtual laboratory concept (also known as I-Labs) to environmental engineering education at MIT. The objective of the project is to develop a network of permanently instrumented boreholes - the 'smart wells', which continuously monitor groundwater conditions over an area of hydrological interest. When coupled with sensors for monitoring external influences such as precipitation and contaminant sources, the 'Smartwells' network provides a rich educational infrastructure. 'Smartwells' combines the benefits of traditional indoor laboratories and field excursions. In addition to real-time data monitoring, this project is also intended as an educational visualization tool for the undergraduate courses offered by the Department of Civil and Environmental Engineering. Students have online access to hydrological data in a quasi-laboratory setting and at the same time have the opportunity to study the complexities of a real-world hydrological system.

1.1 Motivation

In the information age, environmental issues have become of prime concern due to dramatic increase in the pollution levels in all parts of the world. Underlying aquifer flow in environmentally sensitive places plays an important role in characterizing the environmental condition of the place. Ground water flow in such places of hydrologic interest has to be

monitored in order to characterize the groundwater and identify changes or trends in water quality over time. This could help in identifying existing or emerging water quality problems. Information gathered from various data sources regarding the change in water level and quality during various seasons can be used to characterize trends in the physical, chemical and biological condition of the environment. There is thus a pressing need for environmental data monitoring in places of hydrological and environmental interest. Real time analysis of such acquired data will help us address many environmental challenges faced by the industry. Information technology enables us to integrate two systems for continuous data acquisition and analysis and accomplish the task of real-time data monitoring and control. Coupling the emerging sensing technology with an Internet infrastructure can enable efficient data monitoring of conditions over an area of interest.

With the rapid proliferation of networked devices, accelerated by the growth of Internet and wireless communication standards, the next generation in monitoring systems seems to be that of wireless sensor networks. Such advances in sensing technology find very useful applications in Civil Engineering systems. In a sensor rich environment, it is essential to process large chunks of data efficiently and reliably to be able to come to reasonable conclusions about the state of the system. Efficient real-time monitoring coupled with fast data rendering and decision-making capabilities can go a long way in monitoring Civil and Environmental Engineering infrastructure.

1.2 Purpose

The 'Smart Wells' project aims at real-time hydrologic and water table monitoring of wells from a remote location using a combination of wireless sensor network, state-of-the-art sensors for measurement, data acquisition and visual rendering of acquired data on a mobile computer. We are developing a website which allows real-time access to acquired hydrologic data providing a perpetual monitoring capability, the ability to analyze acquired data with visual rendering tools, and a data archiving capability for later studies. This project will be used by students/professors from Hydrology and Environmental Engineering to study groundwater

hydrology in their classrooms and laboratories. The results and source code will be available in the public domain for use by other academic institutions. The project can be deployed in actual field settings with a proper scale-up of the wireless sensor network.

1.3 Layout of the thesis

Chapter 2 discusses the Smartwells project giving an overview of the same. This is followed by the software-hardware interface aspect of data collection, discussing how to interface the measuring equipment with the monitoring server through the Internet. The development environment is described followed by the internals of data-polling from the instrument. The chapter also discusses the sensors used and interfacing the data acquisition hardware to a computer. This is followed by details on how to broadcast data using TCP/IP sockets using the multithreaded DataSocket API available from National Instruments. The later sections deal with archiving real-time data in a database. The database model is discussed along with the JDBC classes used to communicate with back-end SQL based databases. Finally, the processing and visualization of live and archived data is discussed. The client side code for retrieving data from a DataSocket server and techniques to retrieve data from a database are reviewed. Chapter 3 focuses on different distributed development technologies such as DCOM, CORBA and Java/RMI. A distributed application sample is discussed in the context of the Smartwells project. Later, these technologies are compared and their drawbacks are discussed. Emerging technologies such as SOAP are introduced and their advantages on the older technologies are cited. Chapter 4 then deals with the architecture and design of the newer distributed technologies, specifically SOAP. An application sample in the context of Smartwells is again discussed. Chapter 5 then concludes the material presented in the thesis, and details further goals of the work. The appendices have additional details pertaining to the implementation of the software. Appendix A covers the IDL interface, client and server side code for the application sample implemented using DCOM, CORBA and Java/RMI.

CHAPTER 2

SMARTWELLS PROJECT

This chapter starts with an overview of the Smartwells project. It then addresses the remote-monitoring hardware aspects of the project [8, 9, 17-19, 21]. The primary focus is on the sensors used for monitoring aquifers such as the Smartwells. The section on data acquisition systems focuses on the distributed system, *FieldPoint*, manufactured by National Instruments, and the interface between the measuring equipment and the server monitoring via the Internet. The chapter also discusses the issue of interfacing the data acquisition hardware to a computer. The development environment is described, followed by internals of data-polling from the instrument. This is followed by details on how to broadcast data using TCP/IP sockets using the multithreaded *DataSocket* API available from National Instruments. The later sections deal with archiving real-time data in a database. The database model is discussed along with the JDBC classes used to communicate with back-end SQL based databases. Finally, the processing and visualization of live and archived data is discussed. The client side code for retrieving data from a *DataSocket* server and techniques to retrieve data from a database are reviewed.

2.1 Project Overview

The main objective of the Smartwells project was to design and implement a scaleable, real-time system to monitor aquifer hydrology. This would cover the sensing, transmission, archival and rendering aspects of the whole system. The goal was also to experiment with the state of the art in sensing and monitoring, including different hydrological sensors and emerging wireless standards like IEEE 802.11. Then, data obtained by the system was to be made available in real-time as well as in archived format to clients anywhere on the Internet, in a cross-platform manner. This would enable the implementation of distributed information processing and data rendering tools.

The main parameters to be monitored were water level, conductivity in the aquifers and precipitation. Adequate care was taken during design and implementation to ensure that the

developed framework could be easily scaled up to larger, more complex monitoring applications with different hydrological sensors.

The project also aimed at developing educational tools that can enhance the understanding of basic hydrology concepts in classes offered by the Department of Civil and Environmental Engineering and Parsons Lab at MIT. Data Visualization software and simulations can help better comprehend the data monitoring and analysis in various hydrology experiments.

The Smartwells Project is an attempt to monitor the data from an underlying aquifer in real time. The project is divided in to three stages:

- Laboratory Prototype
- Deployment outside the Parsons Lab
- Deployment at a Real-Field site (Waquoit Bay Reserve – Cape Cod)

2.1.1 Laboratory Prototype:

The following figure shows the Laboratory set-up of the Smartwells project. The prototype is set-up in the Design Studio of the Future in Building 1 at MIT. The laboratory set-up served as a test-bed for the various sensors such as the conductivity sensor, water-level sensor and the precipitation sensor. Two Hydraulic Plexiglas cylinders of diameter 15cms and height 75cms were constructed to emulate the wells. The Field Point data acquisition system was used to convert the analog signals from the sensors into digital signals and transmit the digitized data to the data server. The wireless set-up is configured to wirelessly transmit the data to the data server. The same machine *http://smartwells.mit.edu* runs the data acquisition server, database, application as well as the web server.

Laboratory set-up would give necessary inputs for the feasibility of this project and for further real-time deployment.

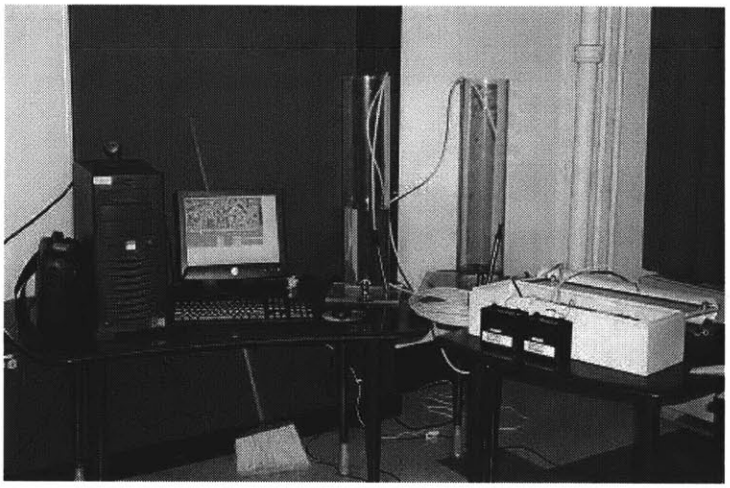

Fig 2.1 Smartwells Laboratory set-up (Room 1-131)

2.1.2. Deployment outside Parsons Lab

Parsons lab is the Environmental Engineering building of MIT. There were three wells present in the parking lot at the side of the building. The prototype as described before would be set up in this building to monitor the aquifer underlying the building. The deployment at this stage would involve installation of the water-level and conductivity sensors in to wells and the rain gauges on the roof of the building. The wireless set-up was decided to be temporarily installed in the third floor copier room of the building and the Field Point Data Acquisition module in the first floor lab adjoining the parking lot. The new IP addresses of the building would be configured for the wireless set-up and the server. This stage had to be deployed by June 2002. Due to construction going on at the Parsons lab, the wells were dug up. Hence this stage of the project is postponed to a future unscheduled date.

The deployment of the project at this stage will be used as an educational aid to the Environmental courses offered by the Civil Engineering Department. Experiments such as salting tests and tracer tests could then be conducted and archived data could be referred to study trends.

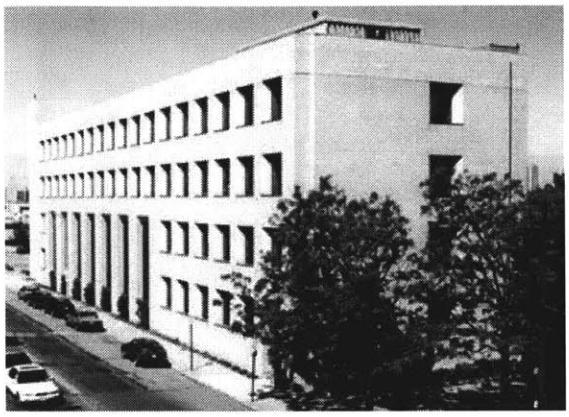

Fig 2.2 Parson's Lab, (Building 48, MIT)

2.1.3. Planned Field Deployment at Waquoit Bay Reserve

The Waquoit Bay National Research Reserve (WBNERR) is located on the south shore of Cape Cod, Massachusetts in the towns of Falmouth and Mashpee. It encompasses some 3000 acres of open waters, barrier beaches, marshlands and uplands. It is around 78 miles from MIT, Cambridge. The ocean waters bring in dynamic changes in the water-levels and the salinity in water due to changes depending on the seasons and tides. The changes in the conductivity also make an interesting study due to the varied mixing of fresh water and sea water.

The proposed implementation at WBNERR would encompass a machine (like the present Smartwells machine) running all the server processes deployed in the main building and the instrumentation equipment installed in a boathouse adjoining the beach. The sensors would be deployed in the soft beach sand using five-foot deep boreholes and would be shielded by Johnson screens to prevent clogging. The sensors would be directly connected to the instrumentation equipment in the boathouse by cables. The instrumentation equipment will then talk to the main server (which sits around 30m further) over wireless LAN. Presently, WBNERR has a temporary dialup access to the internet causing the data to be unavailable online. However, the reserve plans to lease a DSL connection starting June 2002 which will make the Smartwells deployment complete with perpetual access to real-time and archived data.

The later sections go into further details of the implementation of the Smartwells project.

Fig 2.3 Waquoit Bay Reserve, Cape Cod, Massachusetts

2.2 Sensors

A sensor converts a measurable physical quantity from one form to another that can be easily characterized and measured. For instance, a water level sensor converts water heads to voltages or currents that can be easily measured. Calibration is a process by which the sensor is characterized by measuring its response to given known inputs. The calibrated sensor can then be used to quantitatively describe the physical quantity of interest. For example, the voltage output from a calibrated water level sensor can be used to measure the water head.

This section discusses three types of sensors that were used in the Smartwells project, Water Level Sensors (which measure water head), conductivity sensors (which measure the salinity of ground water) and tipping-bucket rain gauges (which measure the precipitation).

2.2.1 Water Level Sensors

A water level sensor is a submersible pressure transducer consisting of a solid state pressure sensor encapsulated in stainless steel submersible housing [Global Water Instrumentation Inc.]. The submersible pressure transducer has a molded-on waterproof cable

which connects the sensor to the monitoring device. The transducer has a two-wire 4-20 mA high level output, five full scales ranges, and is fully temperature and barometric pressure compensated.

Figure 2.4: Global Water WL300 Water Level Sensor

For the Smartwells project, we are using the WL300 Water Level Sensor from Global Water Instrumentation Inc. which provides highly accurate water level measurements for a wide variety of applications in severe environments. The Water Level Sensor has a dynamic temperature compensation system which uses an internal thermister, enabling high accuracy measurements over a wide temperature range. The submersible pressure transducer is easily adapted to the Field Point module from National Instruments. The transducer is easy to install and operate. The Sensor has a two-wire high level 4-20 mA output. Full scale water level ranges are 0-3', 0-15', 0-30', 0-60', 0-120' and 0-250'.

The WL300 submersible pressure transducer is fully encapsulated with marine grade epoxy. The electronics are encased in epoxy so that moisture can never leak in through the O-ring seals or work its way down the vent tube to cause drift or sensor failure (as is the case with other sensors). The vent tube is sealed directly to the sensing element and any moisture that may come down the vent tube will only come in contact with the silicon sensing device and not electronics.

The WL300 submersible pressure transducer uses a unique silicon diaphragm to interface between the water and the sensing equipment. This silicon diaphragm is highly flexible and is in intimate contact with the sensing element, which produces a sensor with exceptional linearity and very low hysteresis. Other metal foil diaphragms tend to crinkle and stretch out over time causing drift, linearity and hysteresis problems. The design of the Water Level Sensor eliminates these problems.

The pressure transducer is available in a 0-3' full scale range which is ideal for measuring shallow flows or small water level changes. The 0-3' range is great for measuring flows in sewers, storm drains, weirs, flumes, lakes, tanks or any water body that is less than 3' deep. The 0-3' sensor accurately measures small changes in water, even when water is only a few inches deep. Other metal foil type sensors typically have serious problems at low level ranges because of crinkling, stretching and drifting.

The Water Level Sensor utilizes a stainless steel micro screen cap to protect the sensing element. This protective cap has hundreds of openings, making it virtually impossible to foul the sensor with silt, mud or sludge.

The WL300 submersible pressure transducer has a two-wire 4-20 mA output signal that is linear with water depth. 10 to 40 VDC is required to run the sensor, so the WL300 transducer can be operated from 12 VDC battery systems. The 4-20 mA signal can run up to 3,000' from the sensor to the logging device. Common twisted pair or electrical extension cord wire may be spliced to the vented cable once the cable is out of the water. The 4-20 mA signal may be converted to 0.5 to 2.5 VDC by dropping the current signal across a 125 ohm resistor.

Specifications:

Pressure Range	0-3', 0-15', 0-30', 0-60', 0-120',0-250'
Linearity and Hysteresis	±0.1% FS
Overall Accuracy	±0.2% (35°F to 70°F)
Overpressure	X4

Burst Pressure	X10
Resolution	Infinitesimal
Outputs	4-20 mA 0.5 to 2.5 VDC across 125 ohms
Supply Voltage	10 to 36 VDC
Response Time	10 mS
Size	3/4" X 8"
Materials	Stainless Steel, Epoxy Silicon
Cable	Polyethylene jacket, 2-conductor, over shield, vented

Table 2.1 Specifications for water level sensor WL300

The submersible pressure transducer may be placed slightly below the lowest expected water level (this is not necessarily the total water depth) and the lowest possible range may be selected to cover the maximum expected water level change.

2.2.2 Conductivity Sensors

The conductivity sensors used in the Smartwells project were WQ301 Conductivity Sensors from Global Water Instrumentation Inc.

The conductivity sensor has two stainless steel electrodes. The outside electrode is a ring and the inside electrode is a wire. The conductivity sensor measures the ability of a solution to conduct an electric current between the two electrodes. The sensor can be used to measure either solution conductivity or total ion concentration of aqueous samples.

The conductivity sensor is automatically temperature compensated using an internal thermister. This means that one sample can be used for measurements in water samples of different temperatures. Without temperature compensation, the conductivity readings would change as the temperature changed, even though the actual ion concentration did not change.

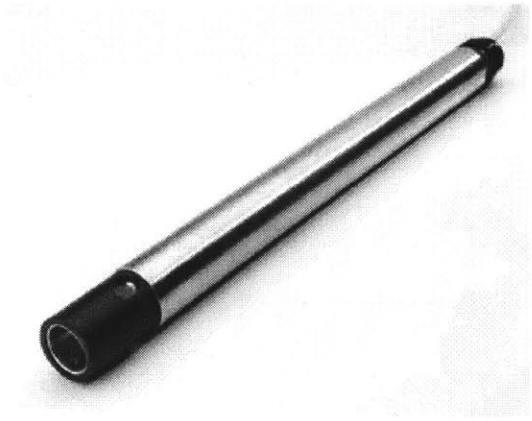

Figure 2.5: Global Water Conductivity Sensor

For the calibration of the conductivity sensor, fill one container with tap water and another with a 5mS solution (where 1 Siemen, the unit of conductivity, is the reciprocal of the resistance in ohms measured between opposite faces of a centimeter cube of an aqueous solution at a specified temperature). Place the conductivity sensor in the latter container; turn on the power supply and the current meter. Let the sensor stabilize for 5 minutes before taking any measurements. Record the output current of the sensor, say X. Remove the sensor and rinse it off with tap water. Fill a contained with distilled water and repeat the above procedure to get an output current, say W. The lower current value for the sensor is equal to W, the output current the sensor would produce if the conductivity were 0. The high current value for the sensor is W, the output current produced if the conductivity is 5 ms. Using the new current values to recalibrate the system which is monitoring the sensor output, we get for some current output Y from the sensor, the corresponding conductivity obtained from the linearity of the sensor is

$$C = 5000 \frac{Y - W}{X - W} \mu S$$

Specifications:

Output	4-20 mA
Range	0 – 5 mS
Accuracy	±1%of full scale

Operating Voltage	12V DC
Current Draw	6.5mA plus sensor output
Warm Up Time	3 seconds minimum
Operating Temperature	-40 C to +55 C
Size of the Probe	1 in diameter and 10.5 in long
Weight	1 lb
Temperature Compensation	2% per C
Electrodes	316 Stainless Steel

Table 2.2 Specifications for conductivity level sensor

2.2.3 Tipping Bucket Rain Gauge

The Tipping Bucket Rain Gauge is a durable low-maintenance weather instrument for monitoring rain rate and total rainfall. It was designed by the National Weather Service to provide a reliable, low-cost tipping bucket rain assessment. Its simplicity of design assures trouble-free operation, yet provides accurate rainfall measurements. For the Smartwells project, we have sourced the rain gauge RG600 from Global Water Instrumentation Inc. which comes with a pulse logger RG700.

The RG600 unit has an 8" orifice and is shipped complete with mounting brackets and 50' of two-conductor cable. The tipping bucket mechanism activates a sealed reed switch that produces a contact closure for each 0.01", 0.2 mm or 1 mm of rainfall. The sensor consists of a gold anodized aluminum collector funnel with a knife-edge that diverts the water to a tipping bucket mechanism. The aluminum housing is covered with white baked enamel. The mechanism is designed so that each tip of the tipping bucket measures 0.2mm or 0.01 in of rainfall. A magnet is attached to the tipping bucket which actuates a magnetic switch as the bucket tips. Thus, a momentary switch closure takes place with each tip of the bucket. The

sensor is connected to an event/pulse counter on an electronic data logger, thereby keeping record of the accumulated rainfall.

The tipping bucket requires a clear and unobstructed mounting location to obtain accurate rainfall readings. The surface should be flat and the environment should be free of vibration. The tipping bucket should be calibrated with the rate of flow of water through the tipping bucket mechanism. At least 36 seconds should be allowed to fill one side of the tipping bucket, representing a maximum flow of 1 inch of rain per hour. If the flow rate is increased, then the unit will read low, since during the last 50% of the tipping time (the time it takes for the bucket to tip), water flows into the empty bucket. Decreasing the rate of flow will not affect the calibration. At flow rates of one inch an hour or less, the water actually drips into the bucket rather than flowing. Under these conditions, the bucket tips between drips and there is no error in the readings.

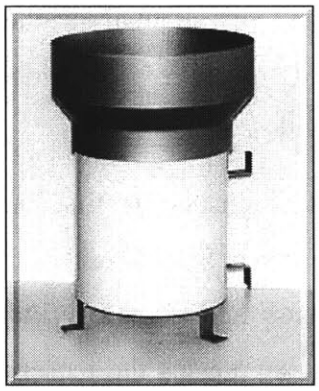

Figure 2.6: Global Water Rain Gauge RG600

Specifications

Resolution	0.01 in
Accuracy	±1% at 1'' per hour
Average switch closure time	135 ms
Maximum Bounce settling time	0.75 ms
Maximum switch rating	30V DC@ 2A, 115V AC @ 1A

Operating Temperature	0 C to +51 C
Size of Gauge	10.125'' x 8''
Weight	2.5 lb
Cable	60', 2 conductor

Table 2.3 Specifications for Rain Gauge RG600

The RG700 is a pulse logger whose output corresponds to the number of tips occurring in the RG600. The RG700 is essentially a capacitive circuit which resets each minute. The amount of rainfall in each minute can be logged corresponding to the number of tips in that minute.

Once sensors are selected for an instrumentation problem, the next issue is to read information from the sensors and process it. This is done using data acquisition hardware, some of which are discussed in this section.

The (analog) signals from the sensors are first usually processed by a signal-conditioning unit, which pre-processes the signal before it reaches the data acquisition hardware. It performs amplification, voltage stabilization and common filtering tasks like noise removal and anti-aliasing. The signal conditioner also powers the sensors whereby a separate power source for the sensors becomes unnecessary.

The conditioned signal is passed to the data acquisition hardware where it is converted from analog to digital by sampling it at a predetermined sampling frequency. The sampled digital output is then fed into the computer. The effectiveness of the data acquisition hardware depends primarily on its *resolution* and *sampling rate*. The resolution determines the number of bits used to represent an analog signal and the sampling rate determines the rate at which the continuous analog signal is discretized.

For the Smartwells project, the data acquisition hardware must be capable of acquiring data, buffering it, and transmitting the data to a central server on request. An integrated signal conditioning unit with data acquisition capabilities and sufficiently high resolution and sampling rates would be ideal. A variety of sensor inputs should be acceptable, the unit should be low-

maintenance and rugged enough for use in harsh environments. The network modules should support protocols.

The *FieldPoint* distributed data acquisition system, manufactured by National Instruments was found to be suitable for the project. It comes with its own high level C library that can be easily interfaced with the feature-laden software development environment from National Instruments, making it very easy to write the data acquisition software.

2.3 Field Point Data Acquisition System

The *FieldPoint* system is a modular distributed I/O system. It allows easy software integration and is one of the most cost-effective instruments available in the market. It is easy to configure, build and maintain reliable distributed I/O solutions. The FieldPoint system includes a variety of isolated analog and digital I/O modules, terminal bases, and network interfaces for an easy connection to open, standard networking technologies.

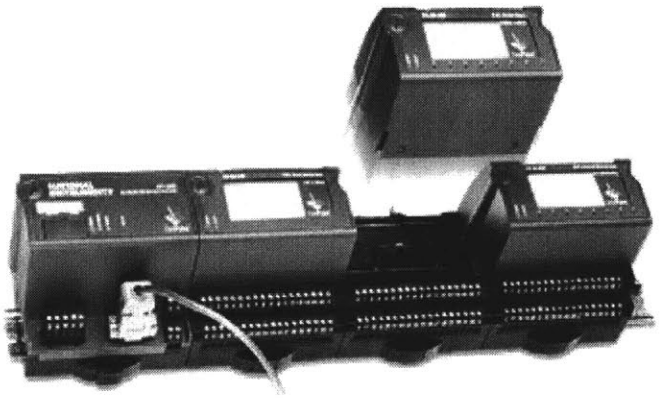

Fig 2.7 Complete FieldPoint Data Acquisition System – National Instruments

The sensor input modules accept a wide variety of sensor outputs at different sampling rates and bit resolutions. The network module then communicates with the host computer using RS 232 or TCP/IP to transfer data. *FieldPoint* supports plug-and-play customization of sensor input modules which makes it modular and easily upgradeable. This chapter discusses the I/O and network modules used in the Smartwells project in detail.

2.3.1 Input-Output (I/O) module:

Two general I/O modules – standard 8/16 channel modules and the dual-channel modules are available with the *FieldPoint* installation. The FP-AI-110 modules support up to eight channels of voltage or current inputs with 16 bit resolution. The FP-AI-110 module is an analog input module with eight analog input channels. The FP-AI-110 is ideal for low frequency signals, and has three configurable filter settings to reject noise. User programmable low-pass filters at 50, 60 and 500 Hz settings are available. Hot plug and play operation, safety isolation, and the 11 input ranges ensure that installation and maintenance are as trouble free as possible.

Fig 2.8 FP-AI-110 I/O Module – National Instruments

Specifications:

Number of channels	8
ADC Resolution	16 bits
Type of ADC	Delta-Sigma
Safety isolation/Working Voltage	250 V rms, designed per IEC 1010 as double insulated

Table 2.4 Specifications for FP-AI-110 I/O Module

The speed of data transfer between the *FieldPoint* module and the host computer depends upon two independent factors, the *sampling rate* of the sensor input module and the *network throughput rate*. The *sampling rate* of the module is defined as the rate at which the ADC (Analog-Digital Converter) in the module digitizes the input and places it in the output register. This is independent of the number of active channels in the module and depends only on the low-pass filter setting. The sampling rates for the FP-AI-110 module are summarized in the following table.

REJECTION FREQUENCY	SAMPLING RATE
50 Hz	1.47 sec
60 Hz	1.23 sec
500 Hz	0.17 sec

Table 2.5: Sampling Rates for FP-AI-110

2.3.2 Network Module:

The network modules communicate with the local I/O module via the high-speed local bus formed by linked terminal bases. The FP-1600 network interface module from National Instruments provides an easy compatible connectivity solution. It connects a node of up to nine *FieldPoint* I/O modules to an Ethernet network and provides up to 100Mb/s data transfer rate.

Fig 2.9 FP-1600 Network Module – National Instruments

The FP-1600 is a bare bones Ethernet module without any onboard memory buffer. It supports both 10 and 100 Mb/s data transfer rates, the actual speed being auto-negotiated depending on the network. Each FP-1600 module can support up to nine sensor input modules.

Specifications:

Network Interface	10BaseT and 100BaseTX Ethernet
Compatibility	IEEE 802.3
Communication Rate	10Mbps, 100Mbps, auto-negotiated
Power Supply Range	11 to 30 volts DC
Power Consumption	7 W + 1.15 (Power for I/O Modules)
Operating Temperature	0 to 55 deg. C.
Dimensions	10.9 by 10.9 by 9.1 cm
Weight	250g

Table 2.6 Specifications for FP-1600 Network Module

The FP-1600 module can be configured using the *FieldPoint* Explorer program available from National Instruments. Configuring the device involves assigning an IP address and configuring the modules attached to it. The *FieldPoint* module and the computer used for the configuration should be on the same class B subnet and have a subnet mask of 255.255.0.0. The configuration can then be saved as an IAK (Industrial Automation Kernel) file, which can be accessed by National Instruments software like Measurement Studio.

The *network throughput rate* is the rate at which the network interface module transfers data between the *FieldPoint* module and the host computer. This depends on a number of factors such as network traffic, total number of channels in the installation (but not on the number of modules itself), *FieldPoint* processing time, etc. The time taken for the network module to read data from the sensor input modules is negligible compared to the sampling rate of the I/O module and the network throughput rate of the network module. The following table shows some typical transfer rates for the FP-1000 module connected to one analog input module, such as FP-AI-110.

	BAUD RATE				
	115.2	**57.6**	**38.4**	**19.2**	**9600 b/s**
1 Channel	6 ms	9 ms	11 ms	19 ms	34 ms
4 Channels	9 ms	12 ms	16 ms	27 ms	49 ms
8 Channels	12 ms	17 ms	22 ms	37 ms	68 ms

Table 2.7 Transfer Rates for FP-1000 [with FP-AI-110 I/O module]

The overall sampling rate is determined by whether the network throughput rate or the sampling rate actually governs.

2.4 Data Collection

The next part of the system involves wireless transmission of the data to the data server wirelessly via a wireless network card, central router and then archiving it in a database. The central wireless router and the wireless network cards use 802.11.b protocol for the wireless

transmission and receiving of data. A data server running LabWindows/CVI collects the data transmitted from the *FieldPoint* module. LabWindows is a data collection and visualization software package created by National Instruments to interface with the *FieldPoint* module. The LabWindows software makes use of the CVI ('C' for virtual Instruments) programming language to collect and display data. Once the data server receives the data, a data socket is written so that other programs have access to it. A Data Sockets Server broadcasts data over TCP/IP sockets using the multithreaded *DataSocket* API. Data sockets are similar to the normal sockets i.e. they are temporary storage locations that package the data transmission and make it available to external computers. While the data is constantly being streamed through on the data socket, a copy of that data is sent to a database and a web server which resides on the same machine for the purpose of the Smartwells project. The client side code for retrieving data from a *DataSocket* server and techniques to write data to the database and retrieve data from it are reviewed later in this chapter. The following section gives us specifications of the wireless technology and devices used.

2.4.1 Wireless Architecture

For the purpose of the Smartwells project, the sensors are physically connected to a data acquisition device in their proximity and a wireless link between the host computer and multiple data acquisition devices is used for data transfer. The data acquisition device has its own networking and processing capabilities in the case of the *FieldPoint* module.

The wireless devices used were off-the-shelf wireless solutions from Orinoco Wireless, Lucent Technologies. These devices use the 2.4-2.485 GHz spectrum for communication and enable data transfer using the IEEE 802.11b (also called WiFi) protocol with data transfer rates of up to 11 Mbps. A *FieldPoint* module and a host computer connected to individual wireless network cards communicate via a central router called the Residential Gateway. While the range of the wireless cards is variable, the residential gateway provides up to 150 m of roaming access in the straight line of sight. In enclosed spaces such as the Design Studio in Building 1 at MIT, this range was found to be around 40m. The last part of the wireless network topology is the Ethernet converter that takes serial or Ethernet inputs and connects to a wireless network card.

This wireless infrastructure was found to be quite feasible for the Smartwells project due to the low sampling rates needed for level, rainfall and conductivity probes. The data flow is from the *FieldPoint* module to the Ethernet converter, then to the central router i.e. the Residential Gateway via the wireless network card and finally to the host computer hosting the data acquisition server via a wireless network card.

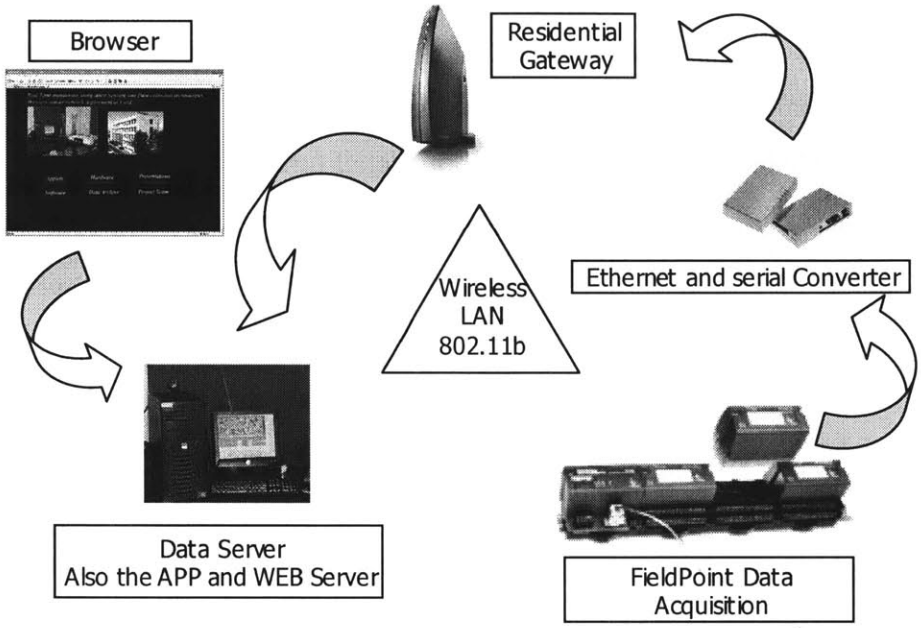

Fig. 2.10 Wireless System Architecture

A Lucent Wireless - Orinoco Residential Gateway (Model RG1000) was used with Orinoco Silver PC cards and 10Base-T Ethernet converter.

2.5 Software for Data Acquisition

The Smartwells project implements a distributed data acquisition and processing system which comprises of the data server, application server and the web server. For the purpose of the Smartwells project, all these parts of the distributed data acquisition architecture sit on the same physical machine *http://smartwells.mit.edu*. Later, we consider the issue of archival of real-time data and retrieval of this archived data.

The *FieldPoint* sensor input modules sample the data from the sensors and communicate it to the network interface module of the installation. The host computer accesses and processes this data by polling the instrument through Ethernet. National Instruments provides a highly compatible software solution 'Measurement Studio' to go with its *FieldPoint* module installations, thereby alleviating the need for socket-level programming. The Measurement Studio software suite comes with LabWindows/CVI, a component which is an ANSI C compliant programming interface.

LabWindows CVI has a convenient interface to *FieldPoint* network modules and comes with significant signal processing capability and provision to spawn off external Java programs. This data acquisition software is discussed in more detail further in this chapter.

Data from the six sensors - two level sensors, two conductivity sensors and two rain gauges is sampled at a low frequency of 1 Hz. The low frequency chosen proves to be sufficient because there is not a substantial change in hydrological data measured by these sensors within this time frame. Also, the full load of processing the acquired data falls on the host computer due to lack of on-board memory buffers in the network module. It is seen that the machine can easily handle the load of database archival and retrieval due to the low sampling frequency chosen. The data acquisition CVI server runs on the same machine as the database (SQL Server 2000), application and web server (Apache on Port 80 and MS-IIS 6.0 with ASP.NET on Port 81). The same machine also hosts the National Instruments *DataSocket* server (which hosts data published by the CVI server) and a separate archival process (which archives this real-time data).

This setup allows an applet hosted on the web server to access both real-time data from the data acquisition server as well as archived data from the database server. The table below lists the services running on the Smartwells server:

SERVICE	DESCRIPTION
`CVI Server(menu.exe)`	Collects data from the *FieldPoint* installation and publishes it to a *DataSocket* server
`DataSocket` `Server (cwdss.exe)`	National Instruments *DataSocket* server

Archival Process (Archive.class)	Archives the real time data from the *DataSocket* Server by writing it to the SQL Server Database
Database Server (MS-SQL Server)	Runs a Database server for data archival and stored procedures to query field data
ASP.NET (with MS-IIS 6.0)	Web server running on port 81 serving ASP.NET pages that access archived data from the SQL
Apache (with Tomcat)	Web server running on port 80 and servlet runner that access real-time and archive data from the

Table 2.8 Services running on the Smartwells server

2.5.1 LabWindows/CVI Interface

The steps involved in data acquisition from the *FieldPoint* module with the help of the LabWindows/CVI server are discussed in this section.

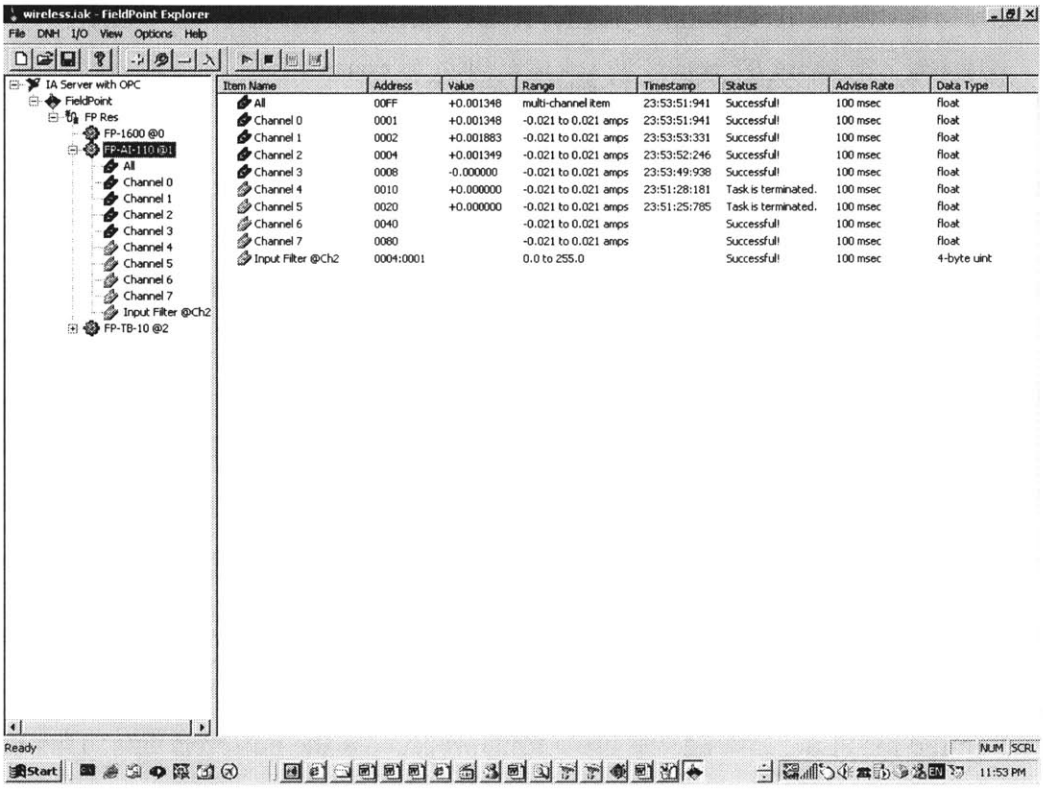

Fig 2.11 FieldPoint Explorer configuration

The *FieldPoint* Explorer program provided by National Instruments is used to configure the *FieldPoint* module and the module configuration is saved in an IAK file. Following this, the instrument libraries are loaded into CVI enabling the data acquisition code to call methods provided by these instrument interfaces. A snapshot of the Field Explorer configuration is presented in Fig.2.11.

In order to talk to the *FieldPoint* module, we get a socket handle to the instrument using the *FP_Open()* function. This functionality is embedded inside the *startMonitoring()* function in the *myfunctions.c* available on the Smartwells website. Next, the *startMonitoring()* function gets a handle over all the channels that need to be monitored using the *FP_CreateTagIOPoint()* call to which we pass the instrument handle, instrument name, channel to monitor and a channel handle as parameters.

```
/* Open a FP connection */
if (status = FP_Open (NULL, &FP_handle)) {
    Error(status);
}

/* Code to create an advise operation for each of the sensor channels */
for(i=0;i<numChannels;i++){
    if(status = FP_CreateTagIOPoint(FP_handle,"FP Res",module[0],itemName[i],&IO_handle[i]))
        Error(status);
    if(status = FP_Advise (FP_handle, IO_handle[i], 100, 0, advisebuf[i],
                           100, 1, NULL, NULL, &advise_ID[i]))
        Error(status);
}
```

The channel handle is then used to poll the instrument using *FP_Advise()* which takes as its parameters the instrument handle, the channel handles, the *advise* rate (instrument polling rate), a global array to hold the channel data, a notify-on-change flag, an array buffer to cache data, buffer size in bytes, *callback* type flag, an optional *callback* function (triggered when the memory buffer gets written to), a *callback* event notifier and a data handle. An interface for editing the *Advise* operators and the module settings is presented in Fig.2.12.

Notify-on-change callbacks may be used for monitoring slow events. But since the Smartwells machine single-handedly runs all the necessary servers, such callbacks may put undue load on the machine. Instead, we use a timer to process the buffered data. This timer calls its callback function after each period and reads data off the memory cache into the data handle. Though a UI timer provided by CVI could have been used due to the low *advise* rates, an asynchronous timer object borrowed from the MIT Flagpole project was used in case some

additional sensors requiring high sampling rates were added on later. This timer makes system-level calls to the OS and works satisfactorily even for high sampling rates.

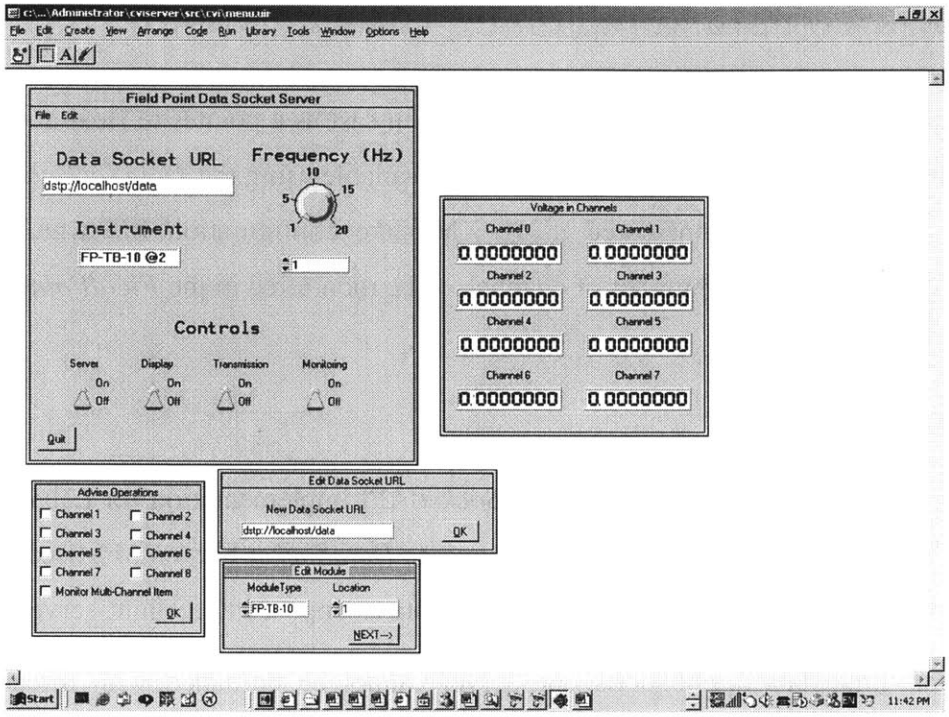

Fig. 2.12 Interface using CVI

The timer frequency is retrieved from the panel (in the example, the frequency is 1Hz) and is followed by the instantiation of an asynchronous timer. The callback function *adviseCB* is triggered after each period. The period is the first argument passed to the asynchronous timer instance.

```
/* initialize async timer */
GetCtrlVal(panelHandle,PANEL_NUMERICKNOB,&frequency);
timerID = NewAsyncTimer (1.00/frequency, -1, 1, adviseCB, 0);
```

Finally, data from the cache is read using the `FP_ReadCache()` method which takes in an instrument handle, an advise operation, a data holder, buffer size and a pointer to a time stamp structure as its parameters.

```
    if(!DEBUG){
        is = FP_ReadCache(FP_handle,advise_ID[i],current_read,BUFFER_SIZE,&dummy);
        value = (float*)(current_read);
        /* ((double)(*value)*scaleFactor[i]-zeroVoltage[i])/sensitivity[i]; */
        channels[i][counter] = (double)(*value);
    }else
        channels[i][counter] = i+rand()/(2*32767.0);
    if(voltpanel)
        SetCtrlVal(voltpanel,VOLTPANEL_CHANNEL_0-i,channels[i][counter]);
}
```

A system timer is used for timing the data that is then cast as a pointer to float and dereferenced to get the final float value. This data is now made available using the *DataSocket* APIs. The data acquisition server can be configured using XML-like configuration file which contains, in addition to other options, the number of channels to be monitored in the *FieldPoint* module.

2.5.2 DataSockets API

The Smartwells project uses the *DataSocket* API implementation for LabWindows / CVI for sharing real-time data. The National Instruments *DataSocket* Server/API uses the publisher-subscriber model (Fig. 2.13) for sharing real-time data as opposed to a client-server one.

In real-time data applications, the server cannot be burdened with thread generation, handling and termination tasks. The *DataSocket* API uses the publisher-subscriber model in which the publisher writes serialized data to a dedicated socket from where it is accessible to all the subscribers. The API takes care of forking multiple connections and uses reflection to enabling dynamic data type recognition of deserialized data at the subscriber end.

To write the data to a *DataSocket* server from LabWindows/CVI, we get a handle to the *DataSocket* URI (in our case, dstp://smartwells.mit.edu/data) and then use the *DS_Open()* function call from the API to post data to this URI. After getting the *DataSocket* URI, the *DS_Open()* function used for posting data takes as its parameters the URI, a connection type, a callback type, optional parameters to pass to the callback function and a handle to the *DataSocket* connection.

```
GetCtrlVal(panelHandle,PANEL_RING,URL);
DS_Open (URL, DSConst_WriteAutoUpdate, DSCallback, NULL, &dsHandle);
```

An automatic update to the server is preferred every time data gets written, and hence the *DS_WriteAutoUpdate* connection mode is selected.

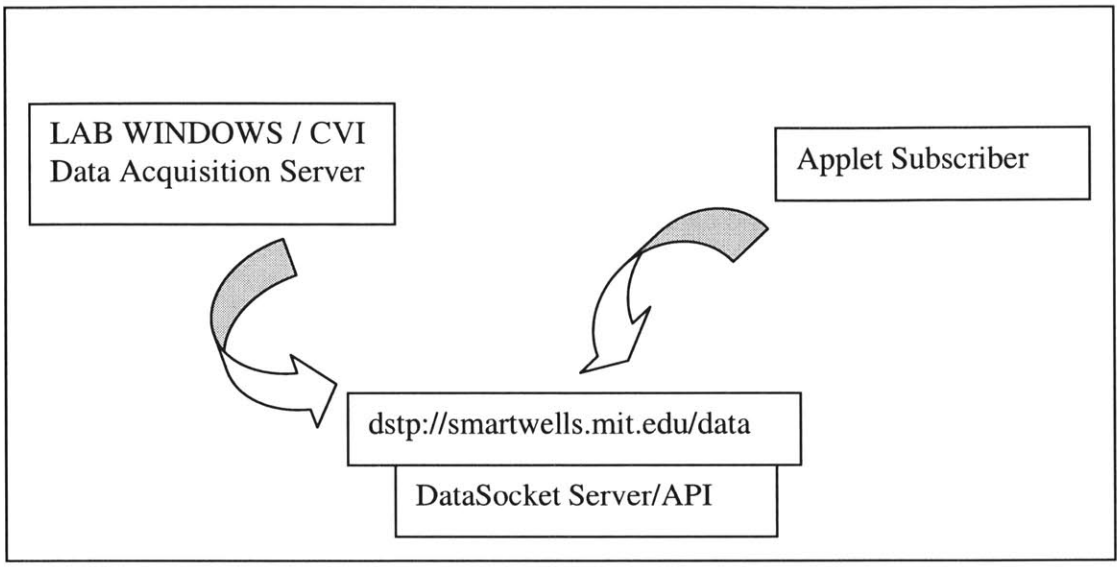

Fig. 2.13: DataSocket Model

A *Callback* function for *DataSocket* events is then triggered with every status change at the server. The *DS_Open()* function then takes an optional parameter that is the data to be passed to the callback function (in our case, *NULL*). The final parameter passed is a handle to the *DataSocket* connection. Upon successful connection, data may be written to the *DataSocket* server via a callback to the asynchronous timer (`adviseCB`). Instead of publishing data at the end of each *advise* operation, data is published in cycles each consisting of 10 *advise* operations on the I/O modules. The Windows time-stamp is converted to a Java time-stamp using the following code:

```
static unsigned long secondsDiff = 2208988800;

switch (event){
    case EVENT_TIMER_TICK:
        GetLocalTime(&dummy);
        localTimeInSeconds = time(NULL);
        localTimeInMillis = (localTimeInSeconds-secondsDiff)*1000.0+dummy.wMilliseconds;
```

The data is written to a *DataSocket* server as a 2D array using a call to the *DS_SetDataValue( )* function.

```
if(counter == 10){
    if(dsHandle){
        hr = DS_SetDataValue (dsHandle, CAVT_DOUBLE|CAVT_ARRAY, channels,numChannels+1,10);
    }

    counter = 0;
}
```

This function takes as its parameters a handle to the *DataSocket* connection, an object type for data written to the server, the 2-D array being eventually written to the server, the number of rows and the number of columns. The data thus published by the *Publisher* can then be accessed by several subscribers by binding the *DataSocket* URI.

2.5.3 Archiving Data

The software framework for archiving the obtained data is now discussed. As mentioned earlier, the data archiving program runs on the web server, which is distinct from the data acquisition server.

The data is archived in a MS-SQL Server 2000 database which runs on the Smartwells machine. The snapshot of the table layout can be seen in Fig. 2.14. The stored procedures run database queries to extract specific well or precipitation data for a required time frame using Transact SQL (for SQL Server Enterprise Manager) / Standard SQL (for Java Archival process) and return datasets of the results.

The data model for the Smartwells database can be seen in Fig.2.15. The database has 3 tables, one each for level, conductivity and rainfall linked together by the instant of data collection as the primary key.

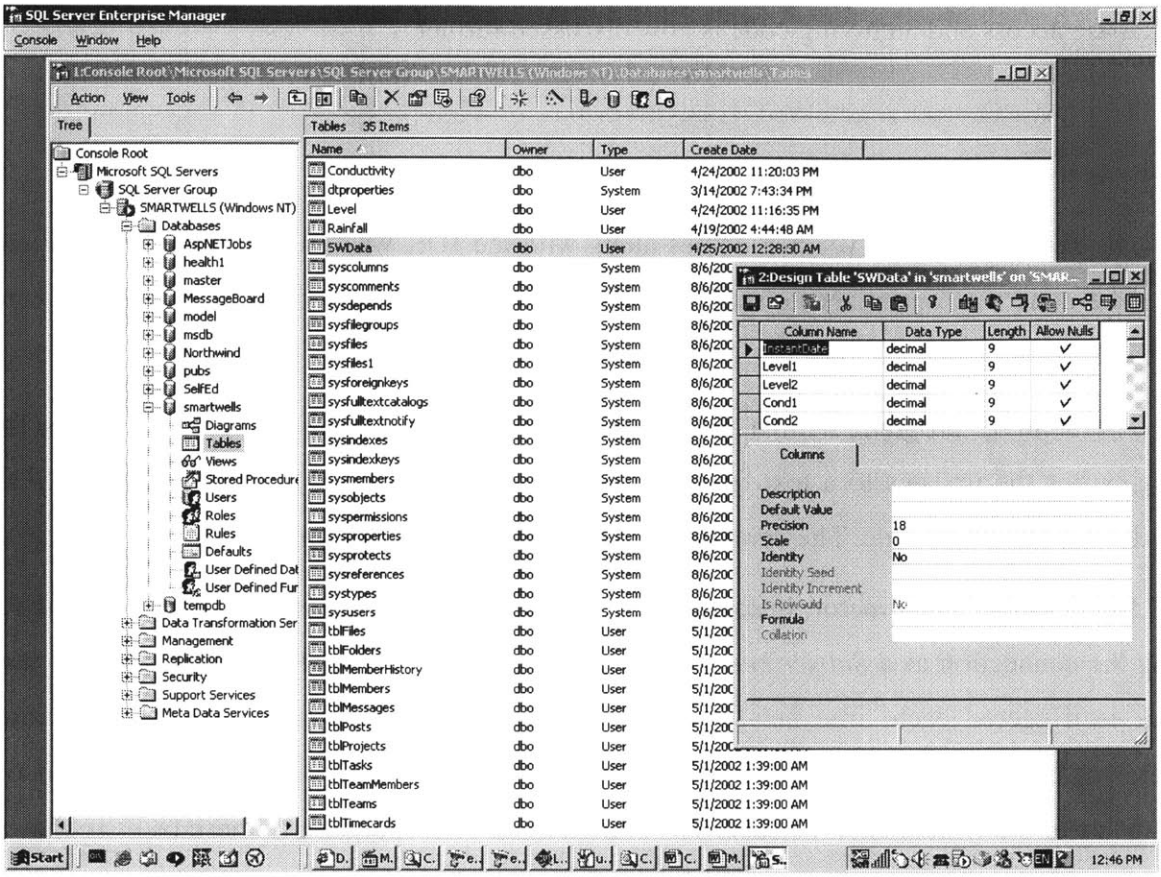

Fig 2.14 Table Layout for MS-SQL Server 2000 Smartwells Database

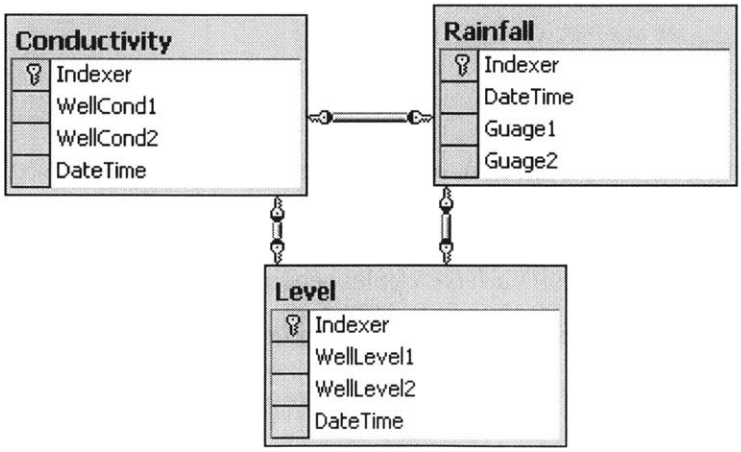

Fig 2.15 Data Model for MS-SQL Server 2000 Smartwells Database

The database access and archiving code is now discussed further.

Database Access and Archival Process

The JDBC-ODBC driver that comes along with the JDK was used for primary database access. The code was optimized to minimize the number of database connections created and persistent connections were ensured.

The archival program written in Java subscribes to the *DataSocket* server, listens for updates, writes the updates to a text file and then does a bulk insert of text data into the SQL Server database every minute. The details of this task are delineated in this section.

The Archive constructor instantiates a *DataSocket*, binds it to the Smartwells URI opening the connection as a *Subscriber*. The access mode is set to *cwdsReadAutoUpdate* so that callback is triggered each time new data becomes available or when the connection status changes. An event listener associated with this *DataSocket* calls the *writeData()* function on every update to the *DataSocket* instance. The code snippet corresponding to these tasks is shown below:

```
ds = new DataSocket();
ds.setURL("dstp://localhost/data");
ds.setAccessMode(DSAccessModes.cwdsReadAutoUpdate);
ds.setAutoConnect(true);
ds.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
        public void DSOnDataUpdate(DSOnDataUpdateEvent event) {
            writeData(event);
        }
    });
```

On each call, the *writeData()* function reads the data from the *DataSocket* as a 2D array of doubles with entries corresponding to each channel monitored and the time stamp. Since the *DataSocket* gets written to after 10 *advise* cycles, each call to *writeData()* gets 10 entries. This data is then written to a *PrintWriter out* as seen in the code:

```
DSData fData = ds.getData();
double readData[][];
try {
    readData = fData.GetValueAsDoubleArray2D();
    for (int i=0;i<10;i++){
        out.println(readData[0][i]+","+
                    readData[1][i]+","+
                    readData[2][i]+","+
                    readData[3][i]+","+
                    readData[4][i]);
    }
```

The *PrintWriter* is flushed and closed each minute and a *bulk insert* is done on the SQL Server database. A new *PrintWriter* is again instantiated as seen:

```
minute = Calendar.getInstance().get(Calendar.MINUTE);
if(minute != currentMinute){
    currentMinute = minute;
    out.flush();
    out.close();
    out = null;
    try{
        stmt = con.createStatement();
        String statement = "BULK INSERT SWData FROM \'"+FILENAME+"\' WITH (FIELDTERMINATOR = \',\')";
        System.out.println(statement);
        stmt.executeUpdate(statement);
    }catch(SQLException e2){
        e2.printStackTrace();
    }

    try{

        out = new PrintWriter(new FileWriter(FILENAME,false), true); // open new buffer
```

A function which starts up and shuts down the *DataSocket* connection is shown in the following code snippet:

```
public void startListening()
{new File(FILENAME).delete();
    ds.connect();
}

public void stopListening()
{
    ds.disconnect();
}
```

2.5.4 Data Visualization

This section describes the applets developed for data visualization and rendering. There are two kinds of data rendered – real-time data and archived data. The real-time data is rendered

using applets which subscribe to the *DataSocket* server to fetch real-time data (just like the Archive program). The archived data is rendered using applets which query the SQL Server database to extract data. Smartwells project also implements a web service to extract archived data using ASP.NET. Chapter 5 details more about this web service and its architecture. Some of the applets developed for the Smartwells project are mentioned below:

The real-time water level monitoring applet subscribes to the *DataSocket* server

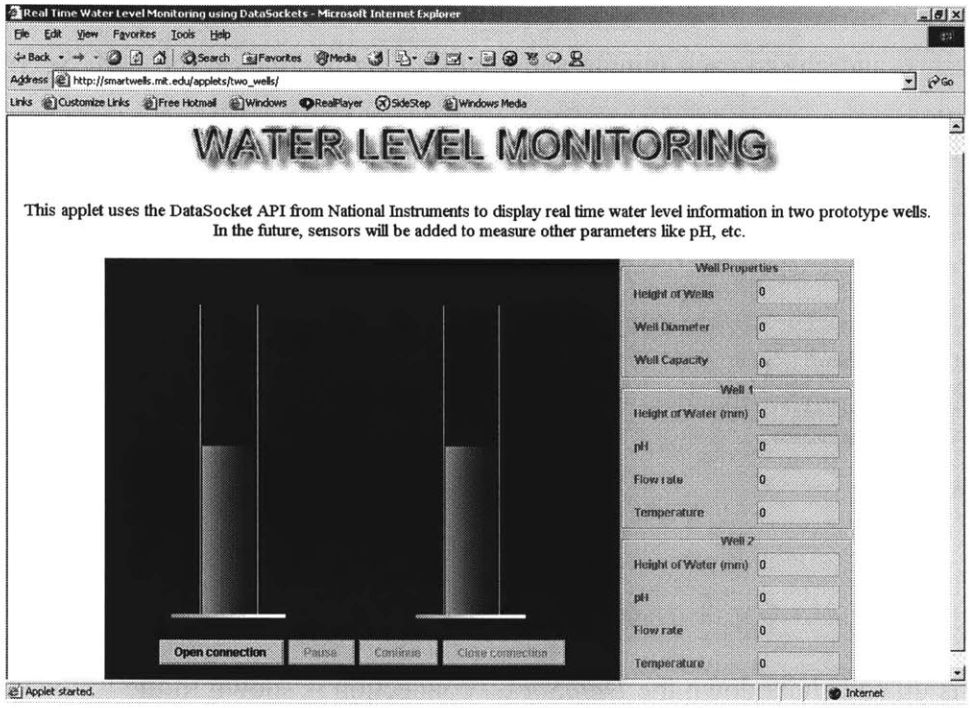

Fig 2.16 Water Level Monitoring Applet

and displays well information (water-level, conductivity, flow rate and temperature out of which the first two sensors are available) for two sample wells being monitored. The well properties in the two wells in the applets can be seen changing in real-time with any change in the actual well properties. The blue level in the applet wells shows the scaled water level in each well so as to give zero water level for an actual 2'' level and full water level for a head of 30''.

The next applet describes the gradient in well properties around the campus by using point values of well properties measured at 10 different measurement points around the campus. The points of measurement are plotted as well points on the background MIT campus map.

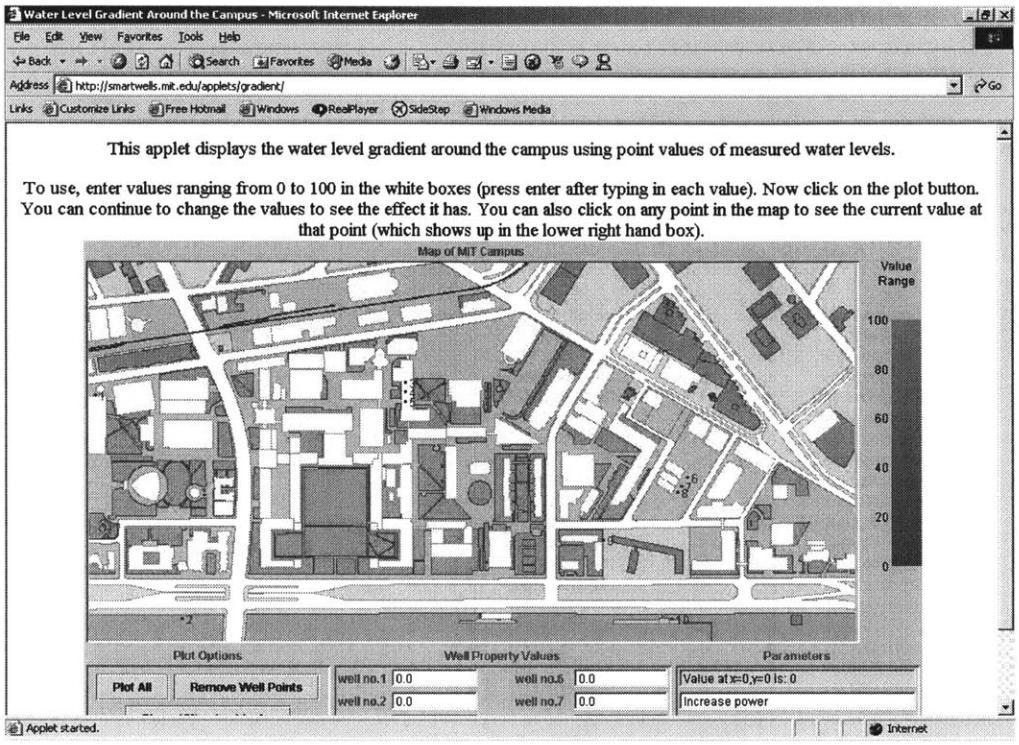

Fig 2.17 Well Properties Gradient Applet

The measurements for the well property values obtained can be displayed in the textboxes. The aquifer properties at any point on the campus can be found out by inverse interpolation (using distances from the measurement spots) of well properties from the measurement spots. The applet can also plot the property gradient circles all around the campus with 10% or 25% cut-off blocks from the points of measurement. Needless to say, this applet also subscribes to the *DataSocket* server to get real-time data.

The next tool considered extracts archived data from the SQL Server database using Microsoft .NET web services. The Smartwells server hosts MS-IIS web server on port 81 and hosts a *SWArchive* web service that contains a *getLevelCond* web method. This method extracts archived data from the SQL Server database using ADO.NET and makes itself available as a

web method which can be invoked remotely by an ASP.NET page as long as the *SWArchive* web service is referenced from the invoking .aspx page. The tool shown below developed using ASP.NET used this method from the *SWArchive* web service and gets the archived level and conductivity data for the two prototype wells.

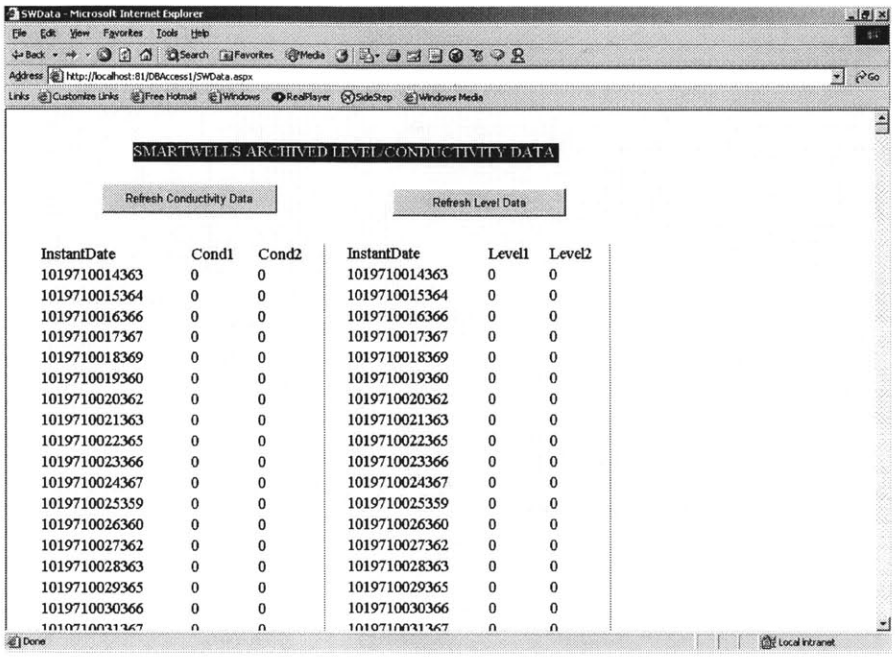

Fig 2.18 Water Level and Conductivity Table Web Service

The Archival program was developed at a later stage in the Smartwells project. Initially, data obtained from the *DataSocket* server was written to text files with the intent of using the LabWindows-SQL Toolkit for the archiving process. Later, the archival program used Java programs with *bulk insert* for the actual archiving due to performance reasons and ease of reuse. The following applets render the conductivity, level and precipitation charts in a bar-chart or line chart format. The comparison charts for water-level, conductivity and precipitation can also be rendered.

The first applet gives the conductivity line chart for the two wells in the given month. The second applet gives the precipitation bar chart for the given month, while the third applet compares water-level, precipitation and conductivity data in a bar chart format.

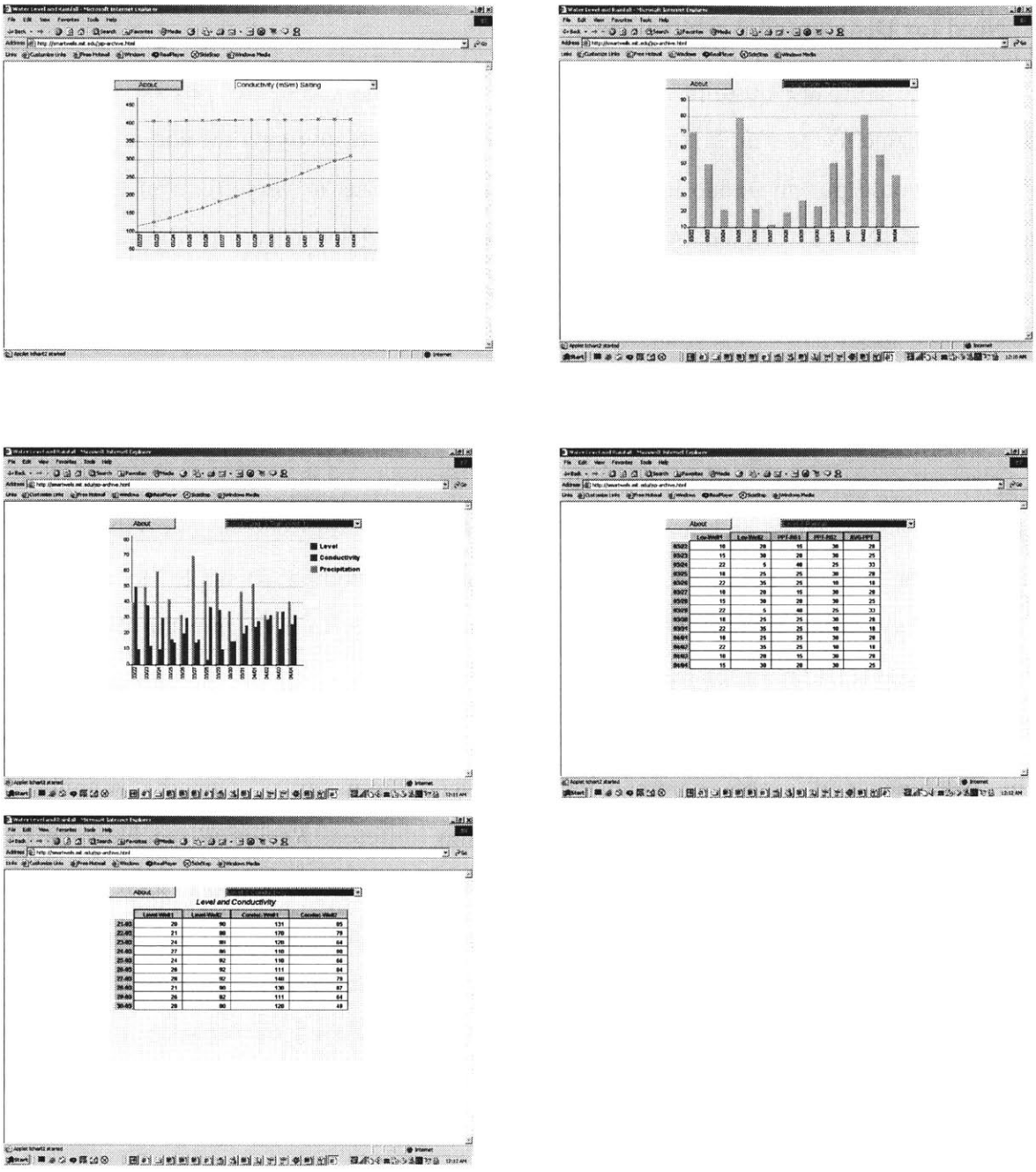

Fig 2.19 Water Level, Conductivity and Precipitation Archived Data Applets

The last two applets present the level and precipitation data (along with the average precipitation) and the level and conductivity data respectively in a tabular format. This concludes the description of the Smartwells project.

2.6 Need for Distributed Architecture

From the software description of the Smartwells project and the various servers the Smartwells machine has to run, it is evident that too many clients invoking the services rendered by the server would put an undue load on the server. This leads to very frequent crashes and unavailability of services offered. Also, giving clients unrestricted access to the database via JDBC-ODBC can be very detrimental to the overall performance of the services offered. A malicious client might affect the integrity of the database by sending it malformed queries. In case of Java, deploying applets that access archived data is very difficult since the data acquisition server and the database server are different. This is due to the sandbox model used by Java applets allowing them to make network connections only to the computer on which it is hosted. Thus in a distributed architecture, an applet could at the very best access only real-time or archived data, but not both. Also, every machine downloading an applet to access the database must have a third party JDBC driver and must be able to configure the data source. This is a very tedious and not user-friendly.

A reasonable solution for this problem would be to deploy the different servers on different machines and enable communication between these server daemons running on different machines. This task can be achieved by using different *Remoting* mechanisms like RPC, Java-RMI, DCOM, CORBA and SOAP. The data acquisition CVI server and the *DataSocket* server can be easily deployed on one machine. The database server can sit on another machine. The most intensive application is the application server which hosts the web services and runs the servlets. The application should therefore be deployed on separate servers. The greater the number of servers used, the better is the application performance and thereby, the quality of service. The services running on the web server will call for the services rendered by the application servers through remote invocation. Fig. 2.20 shows how an applet hosted on the data acquisition server can get access to the database using RMI. The *RMI Server* hosts a *remote* method which it makes available for invocation to client objects via the *RMI Skeleton*. Clients like *Java applets* invoke this remote method from their code as if it were a native method. The call to the remote method and subsequent data retrieval is done via the client-side *RMI Stub*. Here, the client applet invokes a remote method hosted *RMI Server*. This method then queries

the database, extracts the *ResultSet*, packages it into a *Vector* which is serialized and passed back to the client.

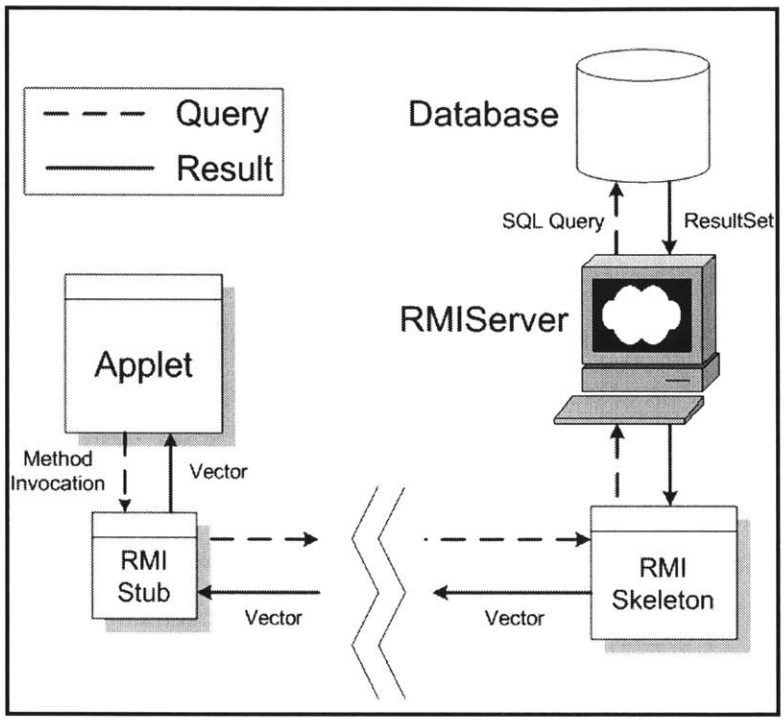

Fig. 2.20 Remote Method Invocation Model

The rest of this work discusses the different *Remoting* mechanisms, their comparison, advantages and limitations. Specifically, we discuss the object oriented remoting mechanisms like RMI, DCOM, CORBA and SOAP. Unlike RPC, all of these are object-oriented, so it is possible to get reference to remote objects themselves instead of just executing remote methods. These mechanisms have their differences. For example, unlike CORBA, Java RMI supports distributed garbage collection, but where CORBA allows remote invocation on almost any type of object, with RMI, it is possible to invoke methods only on remote Java objects. In RMI, objects are serialized (converted into a stream of data) in a Java-specific binary format whereas in SOAP (Simple Object Access Protocol), XML is used for serializing data, allowing it to invoke methods on remote objects written in any language.

CHAPTER 3

DISTRIBUTED DEVELOPMENT – TECHNOLOGY OVERVIEW

3.1 Overview

Today's enterprises no longer live in a protected environment. The whole world is one marketplace. To survive and prosper in this marketplace, enterprises need real-time information about their business processes and operations. They also need to interface with their business partners and share information with them.

Many enterprises already have systems in place for different aspects of their operations. For example, most companies already have systems for accounting, inventory management, production scheduling and customer relationship management. Most of these systems are designed using monolithic proprietary technologies, which are incompatible with each other. Many of these systems are also not very scalable as they were designed before the rapid growth of the World Wide Web. To address this problem, different solution vendors and vendor groups have come out with competing solutions to enable distributed applications. The chief among them are COM+ (From Microsoft), CORBA (From the Object Management Group) and EJB (From Sun Microsystems).

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified whole. These objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application.

Three of the most popular distributed object paradigms are Microsoft's Distributed Component Object Model (DCOM), OMG's Common Object Request Broker Architecture (CORBA) and JavaSoft's Java/Remote Method Invocation (Java/RMI). In this chapter, we will examine the differences between these three models from a programmer's standpoint and an

architectural standpoint [1-7]. At the end of the chapter, we will look at another upcoming technology being pushed mainly by Microsoft and IBM – SOAP [10-16]. Finally, we will have an overview of the chapter thereby enabling better appreciation of the merits and innards of each of the distributed object paradigms [20].

3.2 CORBA

CORBA (Common Object Resource Broker Architecture) relies on a protocol called the **Internet Inter-ORB Protocol (IIOP)** for remoting objects. Everything in the CORBA architecture depends on an **Object Request Broker (ORB)**. The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client. A CORBA object interacts with the ORB either through the ORB interface or through an Object Adapter - either a Basic Object Adapter (BOA) or a Portable Object Adapter (POA). Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform. Major ORB vendors like Inprise have CORBA ORB implementations through their VisiBroker product for Windows, UNIX and mainframe platforms and Iona through their Orbix product.

3.2 DCOM

COM+ is an evolution of an older technology COM (Component Object Model). COM is an object specification, which defines interfaces for objects. Different objects can talk to each other using these interfaces. COM is a language neutral specification and it doesn't matter which

language the objects themselves are coded in as long as they implement the COM interfaces. COM can be implemented on any operating system, though realistically, support on platforms other than Microsoft Windows has been negligible. To facilitate COM objects on different system to talk to each other, the COM specification has been extended to DCOM (Distributed COM), often called *'COM on the wire'*. DCOM supports remoting objects by running on a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer is built on top of RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ *vtable* layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform. Companies like Software AG provide COM service implementations through their EntireX product for UNIX, Linux and mainframe platforms; Digital for the Open VMS platform and Microsoft for Windows and Solaris platforms.

3.4 JAVA/RMI

Java/RMI relies on a protocol called the Java Remote Method Protocol (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface which can be used to access the server object outside of the current Java Virtual Machine (JVM) and on another machine's JVM. The interface exposes a set of methods which are indicative of the services offered by the server object. For a client to locate a server object for the first time, RMI depends on a naming mechanism called an RMI Registry that runs on the Server machine

and holds information about available Server Objects. A Java/RMI client acquires an object reference to a Java/RMI server object by doing a lookup for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page. Since Java/RMI relies on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform. JVM ports are available from many companies including Javasoft and Microsoft.

3.5 Middleware

Middleware component models take a high level approach to building distributed systems. They free the application developer to concentrate on programming only the business logic, while removing the need to write all the 'plumbing' code that is required in any enterprise application development scenario. The enterprise developer no longer needs to write code that handles transactional behavior, security, database connection pooling or threading, because the architecture delegates this task to the server vendor. The competing technologies for middleware are Microsoft MTS (Microsoft Transaction Server), Javasoft's EJB (Enterprise JavaBeans) and OMG's CCM (CORBA Component Model). For the purpose of this thesis, we will not delve deeper into these.

3.6 Application Sample - Smartwells Data Archive Server and Client

The Smartwells Archived Data Retrieval Server functions at the application layer. It extracts archived data from the database server or any alternate back-end and passes it to the application front-end where it may be rendered. The server has a method called *get_rainfall_data( )* to get the rainfall data for the present month.

In this work, Java has been selected as the implementation language for the examples illustrated here for three reasons :

1. Java/RMI can only be implemented using Java.
2. This work compares Java/RMI with other object technologies. Implementing the DCOM and CORBA objects too in Java, lends uniformity to the whole comparison.
3. Java is a very good language to code CORBA and COM objects in that it keeps the implementation simple, easy to understand and very elegant.

Each of these implementations defines an *ISmartwellsData* interface. They expose a *get_rainfall_data ( )* method that returns a float value indicating the average rainfall data for the month passed in. The source code from four set of files is appended in *Appendix-A*. The first set of files is the IDL and Java files that define the interface and its exposed methods. The second set of files show how the client invokes methods on these interfaces by acquiring references to the server object. The third set of files show the Server object implementations. The fourth set of files show the main program implementations that start up the Remote Server objects for CORBA and Java/RMI. No main program implementation is shown for DCOM since the JavaReg program takes up the role of invoking the DCOM Server object on the Server machine. This means you have to also ensure that JavaReg is present on your server machine.

3.7 Implementing the IDL Interface

Whenever a client needs a service from a remote distributed object, it invokes a method implemented by the remote object. The service that the remote distributed object (Server) provides is encapsulated as an object and the remote object's interface is described in an Interface Definition Language (IDL). The interfaces specified in the IDL file serve as a contract between a remote object server and its clients. Clients can thus interact with these remote object servers by invoking methods defined in the IDL.

DCOM

The DCOM IDL file shows that our DCOM server implements a dual interface. COM supports both static and dynamic invocation of objects. It is a bit different than how CORBA does through its Dynamic Invocation Interface (DII) or Java does with Reflection. For the static

invocation to work, the Microsoft IDL (MIDL) compiler creates the proxy and stub code when run on the IDL file. These are registered in the systems registry to allow greater flexibility of their use. This is the *vtable* method of invoking objects. For dynamic invocation to work, COM objects implement an interface called *IDispatch*. As with CORBA or Java/RMI, to allow for dynamic invocation, there has to be some way to describe the object methods and their parameters. Type libraries are files that describe the object, and COM provides interfaces, obtained through the *IDispatch* interface, to query an Object's type library. In COM, an object whose methods are dynamically invoked must be written to support *IDispatch*. This is unlike CORBA where any object can be invoked with DII as long as the object information is in the Implementation Repository. The DCOM IDL file also associates the *ISmartwellsData* interface with an object class *SmartwellsData* as shown in the coclass block. In DCOM, each interface is assigned a Universally Unique IDentifier (UUID) called the Interface ID (IID). Similarly, each object class is assigned a unique UUID called a Class ID (CLSID). COM gives up on multiple inheritance to provide a binary standard for object implementations. Instead of supporting multiple inheritance, COM uses the notion of an object having multiple interfaces to achieve the same purpose. This also allows for some flexible forms of programming.

CORBA

Both CORBA and Java/RMI support multiple inheritance at the IDL or interface level. One difference between CORBA (and Java/RMI) IDLs and COM IDLs is that CORBA (and Java/RMI) can specify exceptions in the IDLs while DCOM does not. In CORBA, the IDL compiler generates type information for each method in an interface and stores it in the Interface Repository *(IR)*. A client can thus query the *IR* to get run-time information about a particular interface and then use that information to create and invoke a method on the remote CORBA server object dynamically through the Dynamic Invocation Interface *(DII)*. Similarly, on the server side, the Dynamic Skeleton Interface *(DSI)* allows a client to invoke an operation of a remote CORBA Server object that has no compile time knowledge of the type of object it is implementing. The CORBA IDL file shows the *SmartwellsData* interface with the *get_rainfall_data()* method. When an IDL compiler compiles this IDL file it generates files for stubs and skeletons.

RMI

Unlike DCOM and CORBA, Java/RMI uses a *.java* file to define its remote interface. This interface will ensure type consistency between the Java/RMI client and the Java/RMI Server Object. Every remotable server object in Java/RMI has to extend the *java.rmi.Remote* class. Similarly, any method that can be remotely invoked in Java/RMI may throw a *java.rmi.RemoteException*. The *java.rmi.RemoteException* class is the superclass of many more RMI specific exception classes. We define an interface called *SmartwellsData* which extends the *java.rmi.Remote* class. The *get_rainfall_data( )* method throws a *java.rmi.RemoteException*.

3.8 Fundamentals of Remoting

To invoke a remote method, the client makes a call to the client proxy. The client side proxy packs the call parameters into a request message and invokes a wire protocol like *IIOP* (in CORBA) or *ORPC* (in DCOM) or *JRMP* (in Java/RMI) to ship the message to the server. At the server side, the wire protocol delivers the message to the server side stub. The server side stub then unpacks the message and calls the actual method on the object. In both CORBA and Java/RMI, the client stub is called the stub or proxy and the server stub is called skeleton. In DCOM, the client stub is referred to as proxy and the server stub is referred to as stub.

3.9 Implementing the Distributed Object Client

DCOM

The DCOM client calls the DCOM server object's methods by first acquiring a pointer to the server object. The *SmartwellsData* DCOM Server object is instantiated. This leads the Microsoft JVM to use the *CLSID* to make a *CoCreateInstance( )* call. The *IUnknown* pointer returned by *CoCreateInstance( )* is then cast to *ISmartwellsData*, as shown below:

ISmartwellsData archdata = (ISmartwellsData) new SmartwellsLib.SmartwellsData();

The cast to *ISmartwellsData* forces the Microsoft JVM to call the DCOM server object's *QueryInterface( )* function to request a pointer to *ISmartwellsData*. If the interface is not supported, a *ClassCastException* is thrown. Reference Counting is handled automatically in Java/COM and the Microsoft JVM takes up the responsibility of calling *IUnknown :: AddRef()* and Java's Garbage Collector automatically calls *IUnknown :: Release( )*. Once the client acquires a valid pointer to the DCOM server object, it calls into its methods as though it were a local object running in the client's address space.

CORBA

The CORBA client first initializes the CORBA ORB by making a call to *ORB.init( )*. Then it instantiates a CORBA server object by binding to a server object's remote reference. Both Inprise's VisiBroker and Iona's Orbix have a *bind( )* method to bind and obtain a server object reference:

SmartwellsData archdata = SmartwellsDataHelper.bind(orb)

Since this is specific to those ORBs, we will use the CORBA Naming Service instead to do the same thing, so that we are compatible with any ORB. We first look up a *NameService* and obtain a CORBA object reference. We use the returned CORBA Object to narrow down to a naming context.

```
NamingContext root =
NamingContextHelper.narrow(orb.resolve_initial_references("NameService") );
```

We now create a *NameComponent* and narrow down to the server object reference by resolving the name in the naming context that was returned to us by the COSNaming (CORBA Object Services - Naming) helper classes.

```
NameComponent[] name = new NameComponent[1] ;
name[0] = new NameComponent("SWRMI","");
SmartwellsData archdata = SmartwellsDataHelper.narrow ( root.resolve (name));
```

Once the client has acquired a valid remote object reference to the CORBA server object, it can call into the server object's methods as if the server object resided in the client's address space.

RMI

The Java/RMI client first installs a security manager before doing any remote calls. This is done by making a call to *System.setSecurityManager()*. It is not mandatory to set a security manager (like JavaSoft's *RMISecurityManager*) for the use of Java/RMI. However, setting a security manager ensures that the Java/RMI client can handle serialized objects for which the client does not have a corresponding class file in its local CLASSPATH. If the security manager is set to the *RMISecurityManager*, the client can download and instantiate class files from the Java/RMI server. This mechanism is actually fairly important to Java/RMI, as it allows the server to generate subclasses for any *Serializable* object and provide the code to handle these subclasses to the client. It is entirely possible to use Java/RMI without setting the security manager, as long as the client has access to definitions for all objects that might be returned. Passing serialized objects is possible only because the JVM provides a portable and secure environment for passing around Java byte code, from which Java objects can be reconstructed at run-time.

The Java/RMI client then instantiates a Java/RMI server object by binding to a server object's remote reference through the call to *Naming.Lookup( )*.

SmartwellsData archdata = (SmartwellsData)Naming.lookup("rmi://localhost/SWRMI");

Once the client has acquired a valid object reference to the Java/RMI server object, it can call into the server object's methods as if the server object resided in the client's address space.

3.10 Implementing the Distributed Object Server

DCOM

In the DCOM server object, all the classes that are required for Java/COM are defined in the com.ms.com package. The DCOM server object shown below implements the ISmartwellsData interface that we defined in our IDL file. The *SmartwellsData* class and the *get_rainfall_data( )* method are declared as public so that they will be accessible from outside the package. Also notice the CLSID specified and declared as private. It is used by COM to instantiate the object through *CoCreateInstance( )* when a DCOM client does a new remotely. The *get_rainfall_data( )* method is capable of throwing a *ComException.*

CORBA

In the CORBA server object, all the classes that are required are defined in the org.omg.CORBA package. The CORBA Server object shown below extends the *_SmartwellsDataImplBase* class that is a skeleton class generated by our CORBA IDL compiler. The *SmartwellsDataImpl* class and the *get_rainfall_data( )* method are declared as public so that they will be accessible from outside the package. The *SmartwellsDataImpl* class implements all the operations declared in our CORBA IDL file. We need to provide a constructor which takes in a name of type String for our CORBA object server class since the name of the CORBA Server class has to be passed on to the *_SmartwellsDataImplBase* class object, so that it can be associated with that name with all the CORBA services.

RMI

In the Java/RMI server object, all the classes that are required for Java/RMI are defined in the *java.rmi* package. The Java/RMI Server object extends the *UnicastRemoteObject* class that

has all of Java/RMI's remoting methods defined and implements the *SmartwellsData* interface. The *SmartwellsDataImpl* class and the *get_rainfall_data( )* method are declared as public so that they will be accessible from outside the package. The *SmartwellsDataImpl* class implements all the operations declared in our Java/RMI interface file. We need to provide a constructor which takes in a name of type *String* for our Java/RMI object server class since the name of the Java/RMI Server class is used to establish a *binding* and associate a public name with this Java/RMI Server Object in the *RMIRegistry*. The *get_rainfall_data( )* method is capable of throwing a *RemoteException* since it is a remotable method.

3.11 The Server Main Programs

CORBA

In the CORBA server, the first thing that has to be done by the main program is to initialize the CORBA ORB using *ORB.init( )*. An *Object Adapter (OA)* sits on top of the ORB, and is responsible for connecting the CORBA server object implementation to the CORBA ORB. Object Adapters provide services like generation and interpretation of object references, method invocation, object activation and deactivation, and mapping object references to implementations. We initialize either the *Basic Object Adapter (BOA)* or the *Portable Object Adapter (POA)* depending on what your ORB supports. In our case, we cite an example using *Inprise's VisiBroker* as the CORBA ORB and hence conform to its implementation requirement where we need to initialize the *BOA*. This is done by calling *orb.BOA_init( )*.

We then create the CORBA server object with the call:

SmartwellsDataImpl SmartwellsDataImpl = new SmartwellsDataImpl ("SWRMI");

We pass in a name *"SWRMI"* by which our object is identified by all CORBA services. We then inform the ORB that the Server Object is ready to receive invocations by the statement:

boa.obj_is_ready(SmartwellsDataImpl);

Since we are using the CORBA Object Service's Naming Service for our clients to connect to us, we will have to bind our server object with a naming service, so that clients would be able to find us. To ensure that our code will work with any CORBA ORB and to facilitate our clients by allowing them to use any ORB's *bind( )* method to connect to the server object we modify the code as follows:

```
org.omg.CORBA.Object object = orb.resolve_initial_references("NameService");
NamingContext root = NamingContextHelper.narrow( object ) ;
NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("SWRMI", "");
root.rebind(name, SmartwellsDataImpl);
```

Next, to ensure that our main program sleeps on a daemon thread and does not fall off and exit, we add:

```
boa.impl_is_ready( );
```

We now enter into an event loop and are in that loop till the main program is shut down.

RMI

In Java/RMI Server main program, the Java/RMI client will first have to install a security manager before doing any remote calls. You do this by making a call to System.setSecurityManager(). We then create the Java/RMI Server object with the call:

```
SmartwellsDataImpl SmartwellsDataImpl = new SmartwellsDataImpl("SWRMI");
```

and remain there till we are shut down.

For the DCOM Server implementation, the Java support in Internet Explorer runs as an in-process server, and in-process servers cannot normally be remoted using the DCOM. However, it is possible to launch a "surrogate" executable in its own process that then loads the in-process server. This surrogate can then be remoted using DCOM, in effect allowing the in-process server to be remoted. You can use JavaReg's /surrogate option to support remote access to a COM class implemented in Java. When first registering the class, specify the /surrogate option on the command line. For example,

javareg /register /class:SmartwellsData /clsid:{FE19E681-508B-11d2-A187-000000000000} /surrogate

This adds a *LocalServer32* key to the registry in addition to the usual *InprocServer32* key. The command line under the *LocalServer32* key specifies *JavaReg* with the /surrogate but without the /register option.

HKEY_CLASSES_ROOT
CLSID
{BC4C0AB3-5A45-11d2-99C5-00A02414C655}
InprocServer32 = msjava.dll
LocalServer32 = javareg /clsid:{BC4C0AB3-5A45-11d2-99C5-00A02414C655} /surrogate

This causes JavaReg to act as the surrogate itself. When a remote client requests services from the COM class implemented using Java, JavaReg is invoked. JavaReg then loads the Java Support in Internet Explorer with the specified Java class. (When distributing the Java program, the installation program must install *JavaReg* along with the Java class.) The *LocalServer32* key can be removed by rerunning *JavaReg* with the */class* option, specifying the same class name, but without the */clsid* or */surrogate* options:

javareg /register /class:SmartwellsData

The DCOM registry file is given in the *Appendix-A*.

3.12 Conclusion

The architectures of CORBA, DCOM and Java/RMI provide mechanisms for transparent invocation and accessing of remote distributed objects. Though the mechanisms that they employ to achieve remoting may be different, the approach each of them take is more or less similar.

DCOM

It supports multiple interfaces for objects and uses the QueryInterface() method to navigate among interfaces. This means that a client proxy dynamically loads multiple server stubs in the remoting layer depending on the number of interfaces being used. Every object implements IUnknown. DCOM uniquely identifies a remote server object through its interface pointer, which serves as the object handle at run-time. It uniquely identifies an interface using the concept of Interface IDs (IID) and uniquely identifies a named implementation of the server object using the concept of Class IDs (CLSID) the mapping of which is found in the registry. The remote server object reference generation is performed on the wire protocol by the Object Exporter

In DCOM, tasks like object registration, skeleton instantiation etc. are either explicitly performed by the server program or handled dynamically by the COM run-time system. DCOM uses the Object Remote Procedure Call (ORPC) as its underlying remoting protocol. When a client object needs to activate a server object, it can do a *CoCreateInstance()*. The responsibility of locating an object implementation and activating it falls on the Service Control Manager (SCM). The client side stub is called a proxy. The server side stub is called stub. All parameters passed between the client and server objects are defined in the Interface Definition file. Hence, depending on what the IDL specifies, parameters are passed either by value or by reference. DCOM attempts to perform distributed garbage collection on the wire by pinging. The DCOM wire protocol uses a pinging mechanism to garbage collect remote server object references. These are encapsulated in the *IOXIDResolver* interface. DCOM allows you to define arbitrarily complex structs, discriminated unions and conformant arrays in IDL and pass these as method

parameters. Complex types that will cross interface boundaries must be declared in the IDL. DCOM can run on any platform as long as there is a COM Service implementation for that platform. Since the specification is at the binary level, diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL can be used to code these objects. Each method call returns a well-defined "flat" structure of type *HRESULT*, whose bit settings encode the return status. For richer exception handling it uses Error Objects (of type *IErrorInfo*), and the server object has to implement the *ISupportErrorInfo* interface.

CORBA

CORBA supports multiple inheritance at the interface level. Every interface inherits from CORBA.Object. CORBA uniquely identifies remote server objects through object references (*objref*), which serve as the object handle at run-time. These object references can be externalized into strings which can then be converted back into an *objref*. CORBA uniquely identifies an interface using the interface name and uniquely identifies a named implementation of the server object by its mapping to a name in the Implementation Repository. The remote server object reference generation is performed on the wire protocol by the Object Adapter. The constructor implicitly performs common tasks like object registration, skeleton instantiation etc.

CORBA uses the Internet Inter-ORB Protocol (IIOP) as its underlying remoting protocol. When a client object needs to activate a server object, it binds to a naming service. The object handle that the client uses is the Object Reference. The mapping of Object Name to its implementation is handled by the Implementation Repository. The type information for methods is held in the Interface Repository. The responsibility of locating an object implementation falls on the Object Request Broker (ORB). The responsibility of locating an object implementation falls on the Object Adapter (OA) - either the Basic Object Adapter (BOA) or the Portable Object Adapter (POA). The client side stub is called a proxy or stub. The server side stub is called a skeleton. When passing parameters between the client and the remote server object, all interface types are passed by reference. All other objects are passed by value including highly complex data types. CORBA does not attempt to perform general-purpose distributed garbage collection.

Complex types that will cross interface boundaries must be declared in the IDL. CORBA will run on any platform as long as there is a CORBA ORB implementation for that platform.

Since this is just a specification, diverse programming languages can be used to code these objects as long as there are ORB libraries you can use to code in that language
Exception handling is taken care of by *Exception* Objects. When a distributed object throws an exception object, the ORB transparently serializes and marshals it across the wire.

RMI

RMI supports multiple inheritance at the interface level. Every server object implements *java.rmi.Remote*. The actual class extended, *java.rmi.UnicastRemoteObject*, is merely a convenience class which calls *UnicastRemoteObject.exportObject(this)* in its constructors and provide *equals( )* and *hashCode( )* methods. RMI uniquely identifies remote server objects with the *ObjID*, which serves as the object handle at run-time. There is a substring such as "[1db35d7f:d32ec5b8d3:-8000, 0]" which is unique to the remote server object. RMI uniquely identifies an interface using the interface name and uniquely identifies a named implementation of the server object by its mapping to a URL in the Registry. The remote server object reference generation is performed by the call to the method *UnicastRemoteObject.exportObject (this).*

The *RMIRegistry* performs common tasks like object registration through the *Naming* class. *UnicastRemoteObject.exportObject (this)* method performs skeleton instantiation and it is implicitly called in the object constructor.

RMI uses the Java Remote Method Protocol (JRMP) as its underlying remoting protocol. When a client object needs a server object reference, it has to do a *lookup( )* on the remote server object's URL name. The object handle that the client uses is the Object Reference. The mapping of object name to its implementation is handled by the *RMIRegistry*. Any type information is held by the object itself which can be queried using Reflection and Introspection. The responsibility of locating an object implementation and activating it falls on the Java Virtual Machine (JVM). The client side stub is called a proxy or stub. The server side stub is called a skeleton. When passing parameters between the client and the remote server object, all objects implementing interfaces extending *java.rmi.Remote* are passed by remote reference. All other

objects are passed by value. RMI attempts to perform distributed garbage collection of remote server objects using the mechanisms bundled in the JVM. Any Serializable Java object can be passed as a parameter across processes. RMI will run on any platform as long as there is a Java Virtual Machine implementation for that platform.

RMI relies heavily on Java Object Serialization, these objects can only be coded in Java. RMI allows throwing exceptions which are then serialized and marshaled across the wire.

3.13 Problems with the older protocols

There are some problems with the older protocols like DCOM, IIOP and RMI/IIOP. The chief among them is that these protocols are incompatible. A DCOM based system cannot talk to an EJB based system, for example. If an enterprise has diverse applications on different platforms, these applications cannot be integrated using the older protocols. Another problem is that these protocols are not firewall friendly. Most firewalls are configured to allow access only traffic only through specific ports, the most popular being the HTTP port 80. The older protocols use different ports, which are blocked by most corporate firewalls. This means that applications residing in different physical locations cannot talk to each other even if they have been built on the same platform.

3.14 Newer Technologies and their Comparison

Distributed Development technologies have passed through a complete life cycle and more mature technologies which try to overcome the problems with the older protocols are emerging. Amongst these, *SOAP* is the hottest buzzword in the new distributed applications development world. The *SOAP* specification is being pushed mainly by Microsoft and IBM. Microsoft has already moved over from DCOM to SOAP in a big way. IBM is also committing to the specification in their future distributed development efforts in a major way.

3.15 Introduction to SOAP

SOAP (Simple Object Access Protocol) has been created to address the problems arising from older protocols discussed before. It is built completely on existing technologies like HTTP and XML. SOAP uses lightweight XML to transmit data between different applications. Since XML is a universal standard, all platforms can access and process the information. SOAP also piggy-rides over HTTP through port 80, so corporate firewalls pose no obstacle. Getting different applications on diverse platforms to talk to each other has been the main focus of software development for a long time. When SOAP becomes generally accepted, a Java application sitting on a Unix box will be able to call methods from a COM object on a Windows server. A client side application on an iMac will be able to access an object served by a mainframe. All these will be transparent and will not require any specific administration.

Though both DCOM and SOAP are originally Microsoft technologies, the architecture of the two is fairly different. Moreover, SOAP has been openly accepted by the Java community because of inherent platform and language independence. Fig. 3.1 displays the difference between DCOM and SOAP.

Here, the interaction between the server and a client are shown for both DCOM and SOAP. The structures look similar, but SOAP gives you greater platform and location independence.

Distributed development technologies may be platform specific and pass data in pre-specified formats. Some of them are language dependent while others like RMI need to be programmed in a specific language like Java. The following table gives a brief comparison of the different distributed development technologies.

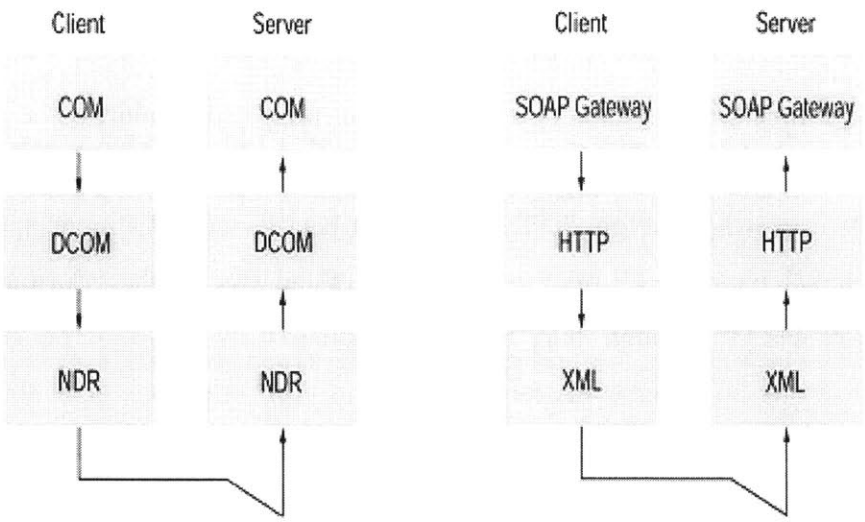

Fig. 3.1 Difference between DCOM and SOAP architecture

Attribute	DCOM	CORBA	RMI	SOAP
Format	Binary	Binary	Binary	XML
Platform	Primarily Windows	Primarily Unix	Independent	Independent
Firewall Friendly	No	No	No	Yes
Programming Language	Independent	Independent	Java	Independent
Access across trusted domains	No	No	No	Yes

Table 3.1 Comparison of DCOM, CORBA, RMI and SOAP

3.16 Conclusion

SOAP is a new technology available for distributed application development professionals. It solves the major problems of platform dependence and language dependence. It is internet ready and is an open standard ratified by the W3C (The World Wide Web Consortium). Implementations of SOAP are available in almost every programming language on almost all the major computing platforms. Existing distributed computing environments are being extended to provide support for SOAP. Microsoft and IBM are moving towards a SOAP

based environment for their respective platforms. Sun has announced something similar with its ONE platform. SOAP has the potential to create a transparent web of services and applications which can be accessed from anywhere by anyone on demand. We will see SOAP in action in the subsequent chapter.

CHAPTER 4

SOAP AND WEB SERVICES ARCHITECTURE

Chapter 4 discusses SOAP in greater detail [13-22]. It addresses the SOAP specification and illustrates the role of SOAP in Web Services architecture with a use-case.

4.1 OVERVIEW

SOAP (Simple Object Access Protocol), to put it simply, allows entirely different objects (for example, Java objects and COM objects) to talk to each other in a distributed, decentralized, Web-based environment. At present, the SOAP specification has been implemented in over 60 languages on over 20 platforms. Suddenly objects everywhere, local and remote, large and small, are able to interoperate. Two very different object types are able to communicate via SOAP.

This chapter introduces SOAP initially in the larger context of Web services, as a protocol that is paired with UDDI (Universal Description, Discovery and Integration) to provide registry and messaging services among businesses. Later, the Web-based underpinnings of the emerging publish-find-bind paradigm are discussed and the mechanisms of SOAP packaging, transport and delivery are revealed.

4.1.1 Evolution of Web Services

Despite all the hype associated with SOAP during its initial introduction, SOAP is simply one component in the emerging picture of the Web as a standards-based, language- and platform-neutral framework for business operations. The reason for SOAP being so important in the entire framework is because of the fact that SOAP is the central component which glues together the entire framework. The business operations utilizing this framework are commonly lumped under the generic tag "Web Services" but Web services themselves are only as good as the infrastructure that supports them. Accordingly, a look at the n-tier architecture of the Internet provides a better picture of the role of SOAP in the context of the framework.

4.1.2 Network Tiers

Three network tiers are evident in the evolution of Web services: TCP/IP, HTTP/HTML, and XML. These tiers build successively on top of each other and remain compatible today.

The first tier, the TCP/IP protocol, is concerned primarily with passing data across the wire in packets. This protocol guarantees transmission across public networks, emphasizing reliability of data transport and physical connectivity. TCP/IP is now the backbone protocol of the Web on which higher-level, standard protocols such as HTTP rely.

The second tier, HTML over HTTP, is a presentation tier and concerns itself with browser-based search, retrieval and sharing of information. The emphasis here is on GUI-based navigation and the manipulation of presentation formats. HTML concerns itself only with the presentation logic and lacks both extensibility and true programming power. Networked desktop environments, burdened with proprietary operating systems and platform dependent software are giving way to the standards-based, open-systems computing of the Internet. In this context, HTML enables sharing hypertext-linked documents in a browser-based environment facilitating communication of text-based information.

A relatively new introduction to the standards-based world is XML, the third and possibly the most compelling tier on the Internet. XML, a strongly-typed data interchange format, provides a new dimension to the HTTP/HTML tier, one in which machine-to-machine communication is made possible through standard interfaces. This layer -- variously described as A2A (application to application), B2B (business to business) or C2C (computer to computer) -- allows programs to exchange data formatted in a platform- and presentation-independent manner. XSLT style sheets may be added as an optional presentation and/or transformational component.

4.1.3 XML: The Key to Describing Web Services

The key to making all this possible is machine-to-machine communication, an area in which XML excels. As syntax for describing data, XML is definition-driven (through the use of DTDs and schemas) and allows information to be manipulated programmatically. This means that most of the guess work can be taken out of B2B communication. Tags can be agreed upon,

interfaces can be defined and processing can be standardized. Web services can then provide for reusable component programs that utilize XML as a standard, extensible communication framework to facilitate this type of computer-to-computer communication.

Web services provide interfaces for the transport of component data and business logic across HTTP. A huge amount of data sits right behind server-side scripts and in legacy repositories, waiting to be accessed by Web browsers or client applications. Web services promise to revitalize corporate software assets now lying dormant in many enterprise domains.

XML plays a crucial role in integrating Web-resident data into enterprise applications and coordinating the business logic that holds these component pieces together. Specific business tasks and services (including workflow logic, business logic, component sequencing logic, transaction logic and so on) can be encapsulated in XML documents and integrated into existing business environments. This allows businesses to leverage existing internal assets and processes and expose this information as Web services, facilitating business transactions and supply chain interaction across the Web. XML is human-readable and text-based, making it ideal as a transport framework for loosely coupled Web services. This facilitates automated transactions which increase productivity, reduce costs and improve services. The presence of standards on the net make automated transactions possible, resulting in productivity gains across the organizations which adhere to these.

SOAP is a technology that derives from an earlier XML-based standard (XML-RPC) and, in some sense, forms the basis for an emerging standard called ebXML (electronic business XML). Work on ebXML is in progress, geared toward providing a comprehensive definition of shared business messages among trading partners. But it is common perception that where SOAP is more modest in scope and less complex in implementation, ebXML is shaping up to be a significantly complex technology which lacks the terse and lucid appeal enjoyed by SOAP.

4.1.4 Loosely Coupled Systems

Web services decouple objects from the platforms that hold them hostage, thus facilitating interactions among platform-independent objects, which are able to access data from anywhere on the Web. As part of the movement away from proprietary platforms, Web services

rely on loose, rather than tight, couplings among Web components. Systems that rely on propriety objects are called tightly coupled because they rely on a well-defined but fragile interface. If any part of the communication between application and service object is disrupted or if the call is not exactly right, unpredictable results may occur.

EDI is an example of a tightly-coupled framework for doing electronic commerce. Loosely coupled systems allow for flexible and dynamic interchange in open, distributed Web environments.

4.1.5 Web Services and CORBA

Standardized network transport protocols, platform-independent programming languages like Java, XML and industry-specific dialects, and open component-based server architectures all contribute to this non-proprietary free-for-all. Web services, with its promise of broad-based application interoperability, promises to be the ultimate glue to make these technologies interact, if not seamlessly, at least without the excess baggage that accompanied previous technologies like CORBA and RMI.

In some sense, Web services represent the second coming of CORBA. But whereas CORBA was an object-oriented, IIOP-based binary communications framework, laden with stubs, skeletons and vendor-specific ORBs, Web services are lightweight, HTTP-based, XML-driven, and completely platform- and language-neutral.

4.1.6 Publish, Bind, and Find Model

A Web services framework consists of a publish-find-bind cycle, whereby service providers make data, content or services available to registered service requesters who consume resources by locating and binding to services. Requesting applications tune themselves to Web services using WSDL (Web Services Description Language), which provides low-level technical information about the service desired, grants applications access to XML Schema information for data encoding, and ensures that the right operations are invoked over the right protocols.

Publish, bind and find mechanisms have their respective counterparts in three separate protocols that make up the Web services network stack: WSDL, SOAP and UDDI

(Universal Description and Discovery Interface). To drill down a little deeper on the CORBA analogy, SOAP plays the role of IIOP in CORBA (or JRMP in RMI). It's the binding mechanism between conversing endpoints. WSDL, on the other hand, plays the role of IDL (Interface Definition Language). In this capacity, WSDL defines Web services as a collection of ports and operations. A WSDL port is analogous to an interface, and a WSDL operation is analogous to a method. WSDL publishes Web service interfaces to parties interested in communicating across heterogeneous platforms.

WSDL, however, goes beyond just being an interface definition language; it also includes constructs that let you describe address and protocol information for the Web services you want to publish. The interesting thing about WSDL is that it describes an abstract interface for Web services while simultaneously allowing one in excruciating detail to bind a Web service to a specific transport mechanism, such as HTTP. By abstracting the interface, WSDL functions as a reusable Web service technology. By binding to a specific transport mechanism, WSDL makes the abstract concrete.

Finally, UDDI acts as a registry for publishing and locating Web services. By exposing service information and binding interfaces in a Web-based registry, UDDI provides a shared directory for businesses and customers to locate one another's Web services.

4.2 Building Web Services with SOAP

SOAP lets you build applications by remotely invoking methods on objects. SOAP removes the requirement that two systems must run on the same platform or be written in the same programming language. Instead of invoking methods through a proprietary binary protocol, a SOAP package uses XML, a text-based syntax for making method calls. All information between the requesting application and the receiving object is sent as tagged data in an XML stream over HTTP. From a Web services point of view, SOAP may be implemented as either a client or a server.

4.2.1 SOAP Clients and Servers

A SOAP client is a program that creates an XML document containing the information needed to invoke remotely a method in a distributed system. SOAP clients could be

a traditional desktop application, a Web server or a server-based application. Messages and requests from SOAP clients are sent over HTTP. As a result, SOAP documents are able to traverse almost any firewall, enabling the exchange of information across divergent platforms.

A SOAP server is a code that listens for SOAP messages and acts as a distributor and interpreter of SOAP documents. External Web services may interact with application servers based on J2EE technology, which process SOAP requests from a variety of clients. SOAP servers ensure that documents received over a HTTP connection are converted to a language that the object at the other end understands. Because all communications are made in the form of XML, objects in one language (say, Java) may communicate via SOAP with objects in any other language (C++, for example). It's the job of the SOAP server to make sure the end points understand the SOAP they are served.

4.2.2 SOAP and Java Technology

According to the SOAP 1.1 specification, SOAP is "a lightweight protocol for exchange of information in a decentralized, distributed environment." SOAP does not mandate a single programming model or define language bindings for a specific programming language. In the context of the Java programming language, it's up to the Java community to define the specific language binding. Java language bindings are now being pursued through the JAX-RPC initiative.

Under a recent Java initiative, the JAX family of technologies focus on enabling the creation of Web services using the familiar JSP and EJB component technologies for the Java platform with Servlets and stateless session beans being most likely used for encapsulating Web services.

4.2.3 A SOAP Use-Case

A typical SOAP use case is shown below:

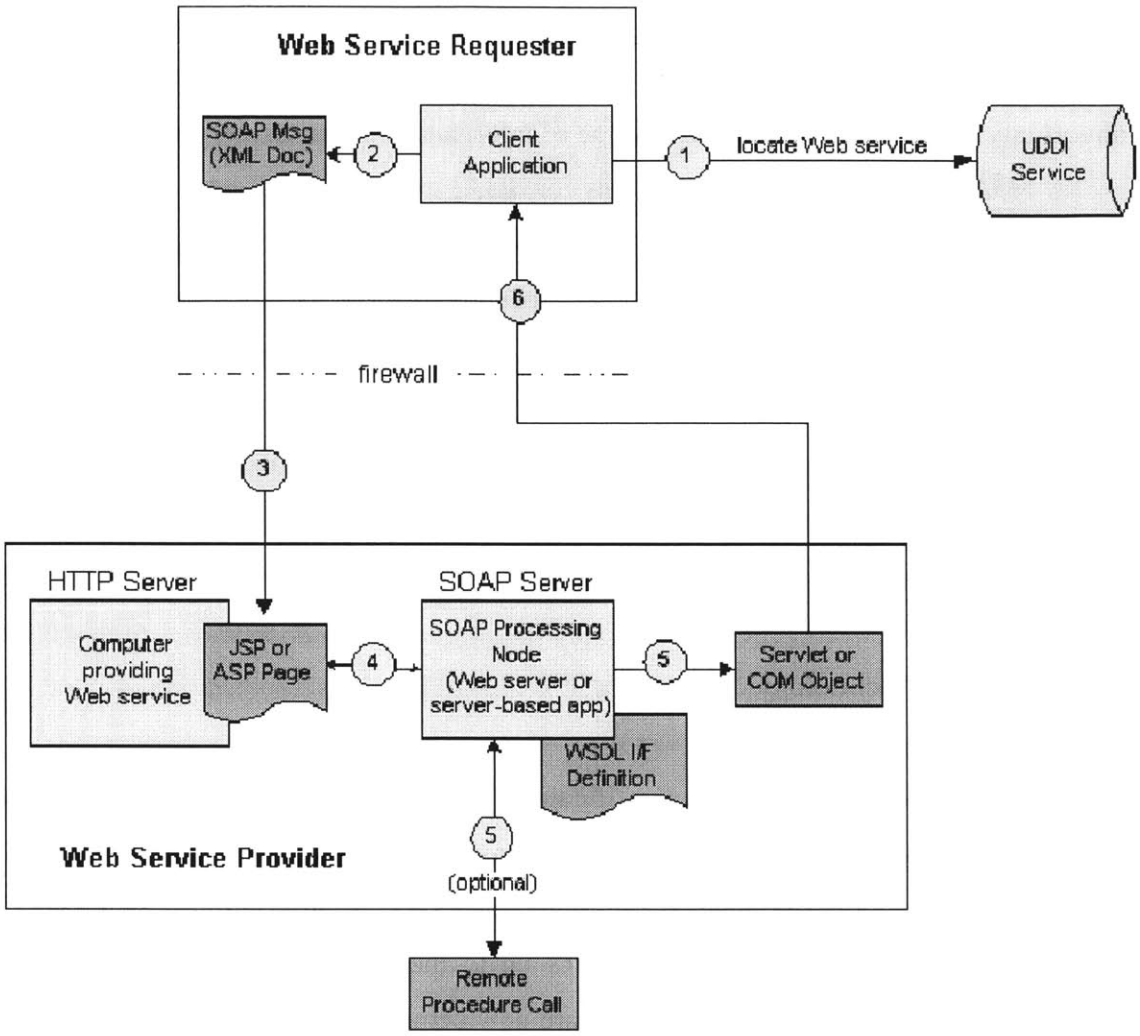

Fig. 4.1 SOAP Use-Case diagram

This section views SOAP from a use-case perspective to grasp the overall processing that takes place in a distributed Web environment. The conceptual backbone of Web services and SOAP can be broadly summarized as follows - A client application somewhere on the Internet consumes Web services. Web services (via SOAP) expose object methods. Object methods access remote data anywhere on the Web. Clients somewhere consume data anywhere on the Web. The case illustrates the following points:

1. A SOAP client uses the UDDI registry to locate a Web service. Rather than manipulate WSDL directly, in most cases a SOAP application will be hardwired to use a

particular type of port and style of binding, and it will dynamically configure the address of the service to be invoked to match the ones discovered through UDDI.

2. The client application builds a SOAP message, which is an XML document capable of performing the desired request/response operation.

3. The client sends the SOAP message to a JSP or ASP page on a Web server listening for SOAP requests.

4. The SOAP server parses the SOAP package and invokes the appropriate method of the object in its domain, passing in the parameters included in the SOAP document. Optionally, intermediate processing nodes may have performed special functions as indicated by SOAP headers prior to receipt of the message by the SOAP server.

5. The request object performs the indicated function and returns data to the SOAP server, which packages the response in a SOAP envelope. The server wraps the SOAP envelope in a response object, such as a servlet or a COM object, which is sent back to the requesting machine.

6. The client receives the object, strips off the SOAP envelope and sends the response document to the program originally requesting it, completing the request/response cycle.

4.3 Role of SOAP in the Web Services Architecture

Having thoroughly set the stage for SOAP and described its crucial role in Web services, we will now delve deeper into the internals of SOAP, its functionality and application.

SOAP is an extensible, text-based framework for enabling communication between objects that have no prior knowledge of each other or of each other's platforms. Client applications can interoperate in loosely-coupled environments to discover and connect dynamically to services without any previous agreements having been established between them. SOAP is extensible, because SOAP clients, servers and the protocol itself can evolve without breaking existing apps. SOAP, moreover, is generous in terms of supporting intermediaries and layered architectures. This means processing nodes can sit on the path a request takes between the client and server. These intermediate nodes process parts of the message specified by SOAP through the use of headers, which allow clients to identify which node works on what part of the message. This type of intermediate header processing is performed by private contract between the client application and the intermediate processing node. SOAP provides a *mustUnderstand*

attribute for headers, which allows the client to specify whether the processing is mandatory or optional. If *mustUnderstand* is set to 1, the server must either perform the intermediate processing specified in the header or throw an exception.

SOAP also defines data encoding rules, called base level encodings or "Section 5" encodings, from the section of the SOAP spec that describes them. SOAP encodings take up most of what constitutes the SOAP 1.1 spec. Without getting bogged down too deeply in XML data type specifics, SOAP encodings can be described briefly as a collection of either simple or compound values.

Simple values are either simple types, like integers, floats and strings, or built-in types as defined in the XML Schema specification. These include data types such as arrays of bytes and enumerations.

Compound values include structures, arrays and complex types as defined by the XML Schema group. Finally, SOAP data encodings specify rules for object serialization; that is, mechanisms for marshaling and unmarshaling data streams across the net.

These base encodings are not mandatory in any way, so clients and servers are free to use different conventions for encoding data as long as they agree on format. This, however, countermands the push SOAP gives to standardized services on the net and is not in the true spirit of SOAP.

Finally, SOAP establishes a set of rules that enable clients and servers to do remote procedure invocation using SOAP as a communications framework. SOAP, which is basically a message-oriented protocol, can, with these conventions, work well as an RPC-type protocol with object serialization as the mechanism that gives SOAP-RPC its wide appeal.

4.3.1 Message Format

SOAP does its messaging in the context of a standardized message format. The primary part of this message has a MIME type of "text/xml" and contains the SOAP envelope. This envelope is an XML document. It contains a header (optional) and a body (mandatory). The body part of the envelope is always intended for the final recipient of the message, while the header entries may target the nodes that perform intermediate processing. Attachments, binary or otherwise, may be appended to the body.

SOAP provides a way for the client to specify which of the intermediate processing nodes has to deal with what header entry. Because headers are orthogonal to the main content of the SOAP message, they're useful in adding information to the message that doesn't effect the processing of the message body.

Headers, for example, may be used to provide digital signatures for a request contained in the body. In this circumstance, an authentication or authorization server could process the header entry -- independent of the body -- stripping out information to validate the signature. Once validated, the rest of the envelope would be passed on to the SOAP server, which would process the body of the message. A closer look at the SOAP envelope will help to clarify the placement and purpose of SOAP header and body elements.

4.3.2 Anatomy of a SOAP Envelope

The SOAP 1.1 spec provides the following sample envelope:

```
<SOAP-ENV: Envelope
 xmlns:
  SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
    <t:Transaction xmlns:t="some-URI">
         SOAP-ENV:mustUnderstand="1"
             5
    </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
    <m:GetAvRainfall xmlns:m="some-URI">
         <month>DEF</month>
    </m: GetAvRainfall>
   </SOAP-ENV:Body>
```

</SOAP-ENV: Envelope>

In this example, a *GetAvRainfall* request is being sent to a Rainfall Web service somewhere on the Web. The request takes a string parameter, a month, and returns a float in the SOAP response.

The SOAP envelope is the top element of the XML document that represents the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers. XML namespaces are used heavily in SOAP to qualify or scope elements in the message to a specific domain.

4.3.3 Namespaces

The first namespace in the example references the SOAP schema, which defines the elements and attributes in the SOAP message. The second namespace refers to SOAP encodings, the base-level data types discussed earlier. Since no additional per-element encoding is specified, this encoding applies to the whole document.

4.3.4 Header

The first element identified in this sample SOAP envelope header is a transaction element, accompanied by a namespace attribute and by the *mustUnderstand* attribute with a value of 1. Since *mustUnderstand* is set to 1, the server accepting this message must perform intermediate processing on this transaction node. You can interpret this to mean that the server and client have previously agreed upon the semantics that govern the processing of this header element, so that the server knows exactly what to do with the contents of the element, in this case 5.

If the server receiving this message doesn't understand the semantics of the transaction header, it is required to reject the request completely and throw a fault. A fault element is a special part of the SOAP body and a well-defined mechanism to ship error information back to the client.

Intermediate processing nodes like this are an example of SOAP's extensibility. Clients include such nodes in a SOAP message to indicate that special processing needs to take place before the contents of the message body can be processed. Ensuring backward

compatibility with existing servers not capable of providing such processing is simply a matter of setting the *mustUnderstand* attribute to 0, which makes the action optional.

In addition to defining transaction nodes like the one described above, a SOAP message may optionally contain header entries specifying nodes that perform authorization processing, encryption, persistence of state, business logic processing and so on. Headers help make SOAP a modular, extensible packaging model. The header processing is entirely independent of the SOAP message body.

4.3.5 Body

The SOAP body in the example contains an XML payload, which does RPC. SOAP is not only a modular packaging model, but also a fairly cryptic packaging model. SOAP does not explicitly show that RPC is being done. In its body, it only exposes a couple of XML elements, one qualified by a namespace. It is up to the SOAP server to understand the document semantics and do the right thing. The server, in effect, provides a framework for dealing with the XML payload in a meaningful way, invoking remote procedure call on the back-end *Smartwells* database to receive the average rainfall for the *month* contained in the message body. All this invocation takes place behind the SOAP RPC curtain.

4.3.6 SOAP-RPC

SOAP messages are fundamentally one-way transmissions from a sender to a receiver, but SOAP messages are often combined to implement request/response mechanisms. RPC using SOAP is based on a few conventions. All request and response messages must be encoded as structures. For each input parameter of an operation, there must be an element (or member of the input structure) with the same name as the parameter. And for every output parameter, there must be an element (or member of the output structure) with a matching name.

A shortened, RPC-based view of the SOAP message presented earlier would look something as follows (Only the body portions of the SOAP request and response envelopes are shown).

Request

```
<SOAP-ENV:Body>
   <m:GetAvRainfall xmlns:m="some-URI">
      <month>DEF</month>
   </m:GetAvRainfall>
</SOAP-ENV:Body>
```

Response

```
<SOAP-ENV:Body>
   <m:GetAvRainfallResponse xmlns:m="some-URI">
      <rainfall >10.50</rainfall>
   </m: GetAvRainfallResponse>
</SOAP-ENV:Body>
```

The request invokes the *GetAvRainfall* method and the response defines a *GetAvRainfallResponse* operation. A convention common to SOAP calls for appending *Response* to the end of a *Request* operation to create a *Response* structure. This output structure contains an element called *rainfall*, which returns the results of the method invocation, presumably as a float.

It's important to note that nowhere in the SOAP envelope are data types explicitly delineated, so we really don't know the type of the *month* or the type of the result parameter *rainfall* just by looking at the SOAP message. Client applications define data types either generically through base-level encodings, or privately via agreed-upon contracts with servers. In either case, these definitions are not explicitly included in the SOAP message.

Finally, in order to do RPC, a lower-level protocol like HTTP is needed. Although the SOAP 1.0 specification mandated the use of HTTP as the transport protocol, SOAP 1.1 1 (and its sister specification "SOAP Message with Attachments") permit the use of FTP, SMTP or even raw TCP/IP sockets. All the serialization and encoding rules general to SOAP apply to RPC parameters as well.

4.4 Summary

SOAP is an XML-based protocol for sending messages and making remote procedure calls in a distributed environment. Using SOAP, data can be serialized without regard to any specific transport protocol, although HTTP is typically the protocol of choice.

SOAP is good for building platform and language-neutral systems that interoperate. Overall, SOAP and Web services account for everything needed to build a distributed application infrastructure on top of XML. SOAP minimizes the problem of multiple-platform incompatibilities in accessing data by resolving the conflict between the COM and Java component object models. Thus, SOAP is the perfect medium for communication between object entities of all types.

CHAPTER 5

FIELD IMPLEMENTATION AND CONCLUSIONS

5.1 Field Implementation at Waquoit Bay

The present research project going on at the Waquoit Bay is the Water Quality Monitoring Program. It involves water quality measurements at 7 sites within Waquoit Bay estuary for the purposes of constructing a long time series of water quality information to determine trends as well as provide a sentinel role to detect changes and events. The Smartwells project would be an ideal implementation for the above mentioned project. There would be real time monitoring and data archiving of the water quality from the various sensors installed.

5.1.1 Proposed Plan of Implementation

The site consists of a maximum 20 foot wide beach with fairly stable water conditions. A small hill adjoining the beach houses the office of the Waquoit Bay Reserve (WBNERR), around 200-250 meters from the beach.

The proposed implementation at WBNERR would encompass a machine (like the present Smartwells machine) running all the server processes deployed in the WBNERR office building and the instrumentation equipment installed in a boathouse adjoining the beach. The sensors would be deployed in the soft beach sand using five-foot deep boreholes and would be shielded by Johnson screens to prevent clogging. The sensors would be directly connected to the instrumentation equipment in the boathouse by cables. The instrumentation equipment will then talk to the main server (which sits around 30m further) over wireless LAN.

The Waquoit Bay Reserve office has a personal network of around 8-10 computers with one node serving as the file server and internet gateway. Presently, WBNERR has a temporary dialup access to the internet causing the data to be unavailable online. However, the reserve plans to lease a DSL connection starting June 2002 which will make the Smartwells deployment complete with perpetual access to real-time and archived data.

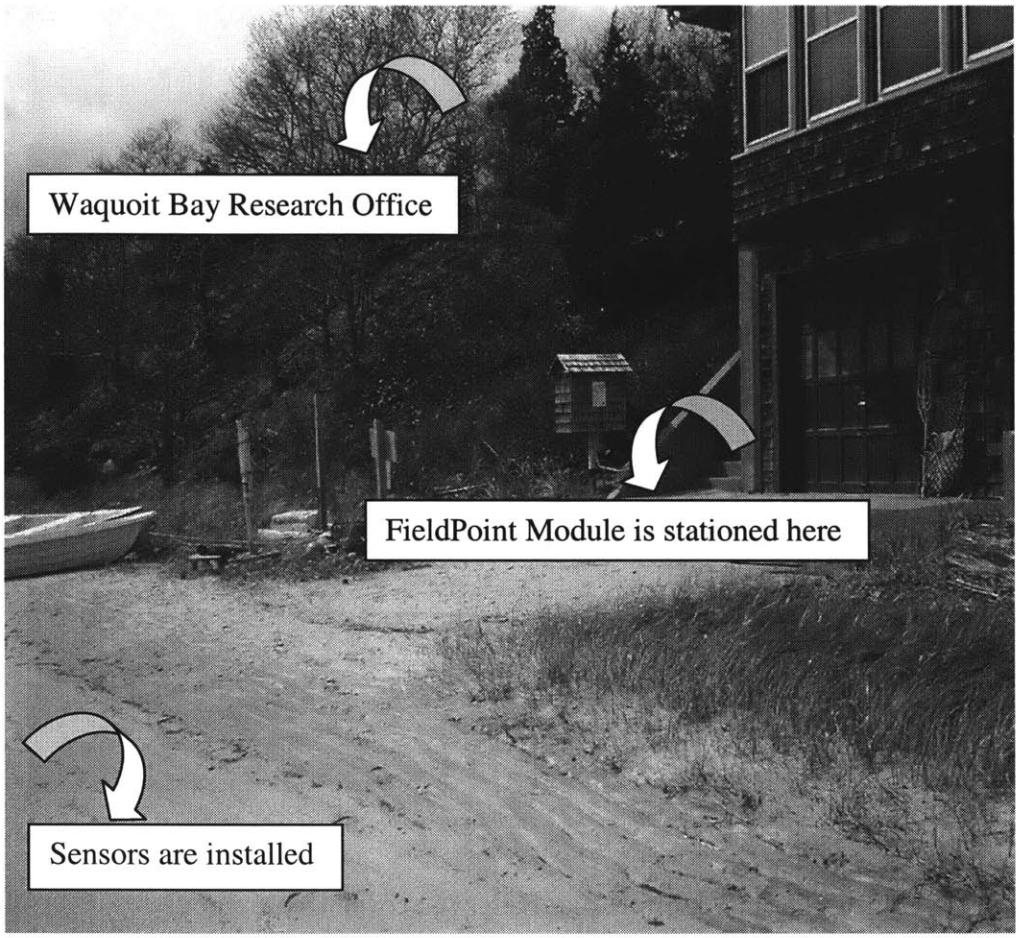

Fig 5.1 Waquoit Bay Sensor Deployment

A temporary implementation however would involve collecting the data and archiving it without Ethernet access. This would mean that the archived data resides on the server and can be accessed online only when the proxy server is connected via the available dial-up connection.

Fig.5.1 shows the deployment of the hardware and sensors on the beachfront at Waquoit Bay. The server deployment at the Waquoit Bay Research office building is shown in Fig. 5.2.

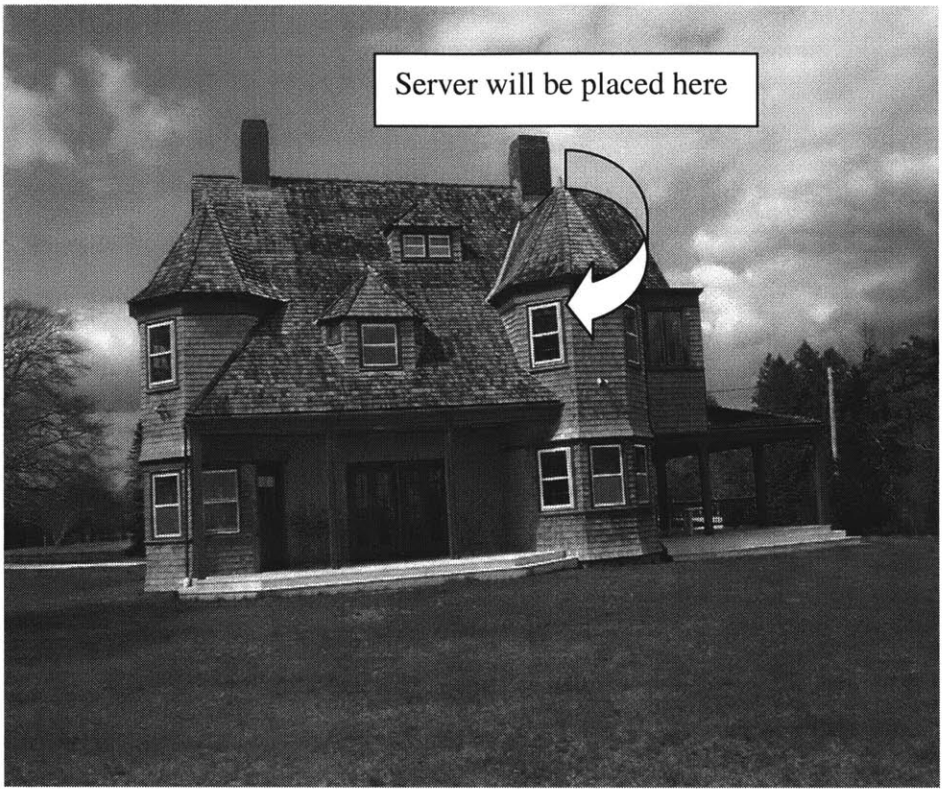

Fig 5.2 Waquoit Bay Reserve Server Deployment

5.2 CONCLUSION

The Smartwells project further strengthens the foundation built by MIT in the realm of real-time data monitoring of physical systems. The project facilitates efficient environmental data monitoring by integrating emerging sensor and wireless technologies. Sensor information gathered regarding the change in water level, quality and precipitation can then be transmitted wirelessly to a distributed infrastructure enabling scalable data analysis and rendering. The distributed development of data monitoring software using technologies like SOAP and RMI enables fast and reliable data processing and also makes the system scalable to include more sensors in the future. Reasonable conclusions could be drawn about the state of the environment in the area of interest.

The Smartwells project has set the stage for further projects involving the possible integration of information technology and environmental engineering studies. The field

implementation of such real-time monitoring and analysis systems as the Smartwells would help resolve environmental issues and monitor pollution.

The project uses the latest available technologies in sensing and wireless transmission. Different options were investigated at each stage in the project and the best available sensors with contemporary technology were selected. The most compatible industry standard for wireless LANs has been chosen keeping in mind our requirements.

The prototype phase for Smartwells project involving integration of various sensors and wireless transmission of the acquired data is complete. The data transmitted wirelessly is archived in a database and real-time monitoring and archived data analysis software has been developed. The next phase of the project would involve real time deployment of the project at the selected site (Waquoit Bay). The present system would be deployed on-site and environmental data would be collected for research and other purposes.

With pressing environmental issues being of prime concern in today's world, MIT is taking the position of a mediator transferring advances in latest technologies from academic institutions and industry to environmental applications in the real world. It is increasingly playing the role of a solutions provider enabling better usage and implementation of available technologies for the purpose of environmental projects.

The overall goal of this thesis is to familiarize an individual with the concept of incorporating information technology into the civil and environmental engineering infrastructure.

References

[1] B2B Application Integration: e-Business-Enable Your Enterprise, *David S. Linthicum,* Jan 2002

[2] Building B2B Applications with XML: A Resource Guide, Michael Fitzgerald

[3] Building Web Solutions with ASP .NET and ADO .NET, *Dino Esposito*, 2002.

[4] Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI, *Steve Graham, Simeon Simeonov et al*, Jan 2002

[5] Developing Java Enterprise Applications, *Asbury Weiner*, Wiley Press 2000

[6] Distributed Enterprise Solutions, Web Cornucopia, June 2001 *G.S. Raj*

[7] Effective Visual Studio.NET, *Wrox Press* 2001.

[8] Field Point Module National Instruments, 15 May 2002 <http://www.ni.com>

[9] Flagpole Project – MIT, 23 February 2002 <http://flagpole.mit.edu>

[10] Implementing B2B Commerce with .NET: A Guide for Programmers and Technical Managers, *Lyn Robison, Jeff Prosise*, Jan 2002.

[11] JavaSoft RMI Specification and Enhancements JDK1.2, May 2000

[12] Microsoft DCOM Specification, Jan 2002, <http://www.microsoft.com>

[13] OMG CORBA Specification (Version 2.3), 1998 <http://www.omg.org>

[14] Professional SQL Server 2000 Programming, *Rob Vieira, Michael Maston*, 2000.

[15] Professional ASP.NET Web Services, *Wrox Press* 2001

[16] Professional ASP XML - *Bill Kropog, Steven Hahn et al*, 2002

[17] Residential Gateway – Orinoco Wireless, Jan 2001 <http://www.orinocowireless.com>

[18] Sensors: Water-level sensors, Conductivity sensors, Rain gauges, Global Water Resources, 15 Jan 2002 <http://www.globalw.com>

[19] Sensor technology and Application to real time monitoring, *David C Greene* (MIT, 2001)

[20] SOAP Specification 1.1, 2002 <http://www.w3.org/TR/SOAP>

[21] Waquoit Bay Reserve Research – WBNERR, April 2002 <http://www.waquoitbayreserve.org>

[22] XML for ASP.NET Developers - *Dan Wahlin*, 2002

APPENDIX A

INTERFACE IMPLEMENTATIONS

DCOM-IDL file

```
[ uuid(7371a240-2e51-11d0-b4c1-444553540000), version(1.0) ]
library SmartwellsLib
{
    importlib("stdole32.tlb");
    [uuid(BC4C0AB0-5A45-11d2-99C5-00A02414C655), dual]
    interface ISmartwellsData: IDispatch
    {
        HRESULT get_rainfall_data([in] BSTR month, [out, retval] float * rain);
    }
    [uuid(BC4C0AB3-5A45-11d2-99C5-00A02414C655), ]
    coclass SmartwellsData
    {
        interface ISmartwellsData;
    };
};
```

CORBA-IDL file

```
module SmartwellsLib
{
  interface SmartwellsData
  {
        float get_rainfall_data( in string month );
  };
};
```

RMI-Interface Definition

```
package SmartwellsLib;
import java.rmi.*;
import java.util.*;

public interface SmartwellsData extends java.rmi.Remote
{
    float get_rainfall_data( String month ) throws
RemoteException;
}
```

CLIENT IMPLEMENTATIONS

DCOM Client

```
//
// SmartwellsDataClient.java
// SmartwellsDataClient - DCOM Client Implementation
//
//
import SmartwellsLib.*;

public class SmartwellsDataClient
{
 public static void main(String[] args)
 {
  try
   {
       ISmartwellsData archdata = (ISmartwellsData)new SmartwellsLib.SmartwellsData();
       System.out.println( "The average rainfall for the month is " + archdata.get_rainfall_data("MY_MONTH") );
   }
  catch (com.ms.com.ComFailException e)
   {
       System.out.println( "COM Exception:" );
       System.out.println( e.getHResult() );
       System.out.println( e.getMessage() );
   }
 }
}
```

CORBA Client

```
//
// SmartwellsDataClient.java
// SmartwellsDataClient - CORBA Implementation
//
//
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import SmartwellsLib.*;

public class SmartwellsDataClient
{
    public static void main(String[] args)
    {
        try
            {
                ORB orb = ORB.init();
                NamingContext root = NamingContextHelper.narrow( orb.resolve_initial_references("NameService") );
                NameComponent[] name = new NameComponent[1] ;
                name[0] = new NameComponent("SWRMI","");

                SmartwellsData archdata = SmartwellsDataHelper.narrow(root.resolve(name));
                System.out.println("Average Rainfall for given month is " + archdata.get_rainfall_data("MY_MONTH"));
            }
        catch( SystemException e )
            {
                System.err.println( e );
            }
    }
}
```

RMI Client

```
//
//
// SmartwellsDataClient
//
//
import java.rmi.*;
import java.rmi.registry.*;
import SmartwellsLib.*;

public class SmartwellsDataClient
{
public static void main(String[] args)throws Exception
{
if(System.getSecurityManager() == null)
{
System.setSecurityManager(new RMISecurityManager());
}
SmartwellsData archdata = (SmartwellsData)Naming.lookup("rmi://localhost/SWRMI");
System.out.println( "The average rainfall for the month is "
+ archdata.get_rainfall_data("MY_MONTH") );
}
}
```

SERVER IMPLEMENTATIONS

DCOM Server

```
// SmartwellsDataServer
//
//

import com.ms.com.*;
import SmartwellsLib.*;

public class SmartwellsData implements ISmartwellsData
{
 private static final String CLSID = "BC4C0AB3-5A45-11d2-99C5-00A02414C655";

 public float get_conductivity_data( String month )
 {
     float cond[31];

     for( int i = 0; i < month.ndays(); i++ )
         {
             cond += (int) month.charAt(i);
         }

     cond /= month.ndays();
     return cond;
 }

}
```

CORBA Server

```
// I
//
// SmartwellsDataServer
//
//

import org.omg.CORBA.*;
import SmartwellsLib.*;

public class SmartwellsDataImpl extends _SmartwellsDataImplBase
{

    public float get_rainfall_data( String month )
    {
        float av_rain = 0;
        // Query DB and extract rainfall values for the month in rain[]
        for(int i = 0; i < month.ndays(); i++)
            {
                av_rain +=  rain[i];
            }
        av_rain /= month.ndays();
        return av_rain;
    }

    public SmartwellsDataImpl( String name )
    {
        super( name );
    }


}
```

RMI Server

SWRMI-Interface

```
/*
 * SWRMI.java
 *
 * Created on April 21, 2002, 9:37 PM
 */

import java.rmi.*;


/** Remote interface.
 *
 * @author kashish
 * @version 1.0
 */
public interface SWRMI extends java.rmi.Remote {

    public float| get_rainfall_data(string month) throws java.rmi.RemoteException;

}
```

RMI Server

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;


public class SWRMIServer
{

  public static void main(String[] args) throws Exception
  {
      if(System.getSecurityManager() == null)
      {
       System.setSecurityManager( new RMISecurityManager() );
      }
      SWRMIImpl myObject = new SWRMIImpl( "SWRMI" );
      System.out.println( "RMI Smartwells Server ready..." );
  }
}
```

SWRMI – Interface Implementation

```
/*
 * SWRMIImpl.java
 *
 * Created on April 21, 2002, 9:37 PM
 */

import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

/** Remote object implementing the SWRMI interface.
 * It can be exported for example by java.rmi.server.UnicastRemoteObject.exportObject()
 * @author kashish
 * @version 1.0
 */
public class SWRMIImpl extends java.lang.Object implements SWRMI {

    /** Constructs SWRMIImpl object.
     */
    public SWRMIImpl(String name) {
    try
    {
      Naming.rebind( name, this );
    }
    catch( Exception e )
    {
      System.out.println( e );
    }
    }

    public float get_rainfall_data(string month) throws java.rmi.RemoteException {

      float av_rain = 0;
      // Query DB and extract rainfall values for the month in rain[]
      for(int i = 0; i < month.ndays(); i++)
          {
            av_rain +=  rain[i];
          }
      av_rain /= month.ndays();
      return av_rain;

    }

}
```


501 - 50