A Web-Based Virtual Laboratory for Monitoring Physical Infrastructure

by

Raghunathan Sudarshan

Bachelor of Technology, Civil Engineering (2000)
Indian Institute of Technology Madras

Submitted to the Department of Civil and Environmental Engineering in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Civil and Environmental Engineering

at the

Massachusetts Institute of Technology

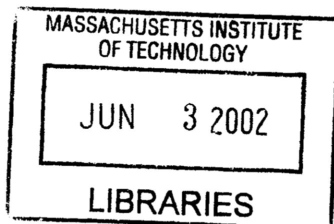
June 2002

© Massachusetts Institute of Technology
All rights reserved

Signature of Author ....
Department of Civil and Environmental Engineering
May 24th 2002

Certified by ....
Kevin Amaratunga
Assistant Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by ....
Oral Buyukozturk
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 3 2002
LIBRARIES

A Web-Based Virtual Laboratory for Monitoring Physical Infrastructure

by

Raghunathan Sudarshan

Submitted to the Department of Civil and Environmental Engineering on May 24th, 2002 in partial fulfillment of the requirements for the degree of Master of Science in Civil and Environmental Engineering

ABSTRACT

In this thesis, the design and implementation of a virtual laboratory for monitoring real-world physical infrastructure over the Internet are described. In particular, the implementation of such a remote laboratory to monitor the deformation under lateral wind loading of a flagpole at the MIT campus is considered. This project is one of several web-accessible remote laboratories that are part of Project I-Campus at MIT.

In the first chapter, the motivation, objectives and educational benefits of such a monitoring laboratory are described. Remote virtual laboratories such as the one discussed in this thesis can provide an effective means for illustrating concepts in structural dynamics and signal processing and can be used to expose students to advances in sensor technology. They can also serve as ideal design and test platforms for active control algorithms. In the next chapter, a brief overview of the hardware aspects of this remote monitoring project, namely the sensors, data acquisition units and wireless-networking components is given. The next three chapters discuss the three main software components of this effort: programs that collect and disseminate data, those that archive the data and clients that process real-time and archived data in many ways.

The sixth chapter describes some of the educational tools and simulations that have been developed as part of this virtual laboratory and discusses how they can be used in a classroom setting. The final chapter summarizes the thesis and provides directions for further research.

Thesis Supervisor: Prof. Kevin Amaratunga

Title: Assistant Professor of Civil and Environmental Engineering

ACKNOWLEGEMENTS

I would first like to thank my advisor, Prof. Kevin Amaratunga for abundant and insightful guidance during the course of this project.

I would then like to express my appreciation to Microsoft Corporation for funding this research as part of Project I-Campus at MIT.

Thanks are also due to all the students who worked on all aspects of this project during the last two years.

Finally, I would like to thank my family for being a source of strength and encouragement throughout my graduate study.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION 10

1.1 MOTIVATION 10
1.2 OBJECTIVES 12
1.3 RELATION TO OTHER I-LAB PROJECTS 13
1.4 ROADMAP OF THE THESIS 14

CHAPTER 2. SENSORS AND DATA ACQUISITION HARDWARE 15

2.1 OUTLINE 15
2.2 SENSORS 15
2.3 DATA ACQUISITION 19
2.4 WIRELESS SENSOR NETWORKS 26
2.5 INSTALLATION OF SENSORS ON THE FLAGPOLE 28

CHAPTER 3. DATA COLLECTION AND DISSEMINATION 29

3.1 SOFTWARE FOR DATA ACQUISITION 29
3.2 INSTRUMENT INTERFACE WITH LABWINDOWS/CVI 31
3.3 DATA DISSEMINATION USING DATASOCKETS 35
3.4 CONFIGURING THE DATA ACQUISITION SERVER 40

CHAPTER 4. DATA ARCHIVAL 41

4.1 SOFTWARE FOR DATA ARCHIVAL 41
4.2 THE FLAGPOLE DATA MODEL 41
4.3 DATABASE ACCESS USING JDBC 43
4.4 THE ARCHIVAL PROGRAM 43
4.5 MAINTAINING RELIABLE OPERATIONS 47
4.6 CUSTOMIZING THE ARCHIVAL PROGRAM 49

CHAPTER 5. DATA RETRIEVAL AND VISUALIZATION 50

5.1 SOFTWARE FOR DATA RETRIEVAL 50
5.2 ACQUIRING REAL-TIME DATA USING LABWINDOWS/CVI 50
5.3 DISPLAYING REAL-TIME DATA IN JAVA 53
5.4 ACQUIRING ARCHIVED DATA USING RMI AND JDBC 55
5.5 INTERFACING A SERVLET TO THE DATABASE 62

CHAPTER 6. EDUCATIONAL TOOLS FOR THE VIRTUAL LABORATORY 64

6.1 EDUCATIONAL GOALS 64
6.2 APPLETS THAT UTILIZE REAL-TIME DATA 64
6.3 SIMULATION APPLETS 66
6.4 USING THE SOFTWARE TOOLS IN AN ENGINEERING CURRICULUM.... 69

CHAPTER 7. CONCLUSIONS AND FURTHER WORK 70

7.1 SUMMARY AND CONCLUSIONS 70
7.2 FURTHER WORK 71

APPENDIX A. REFERENCES 73

LIST OF FIGURES

FIGURE 1: ILLUSTRATION OF THE I-CITY CONCEPT 11
FIGURE 2: LOCATION OF THE FLAGPOLE IN THE DUPONT COURT (MARKED WITH A CROSS) .. 12
FIGURE 3: A SIMPLIFIED MODEL OF AN ACCELEROMETER 16
FIGURE 4: ILLUSTRATION OF BULK MICRO MACHINING ([DOSCHER, J., (1995)]). 17
FIGURE 5: A CAPACITATIVE BULK MICRO MACHINED ACCELEROMETER [CROSSBOW TECHNOLOGY, (2000)] 18
FIGURE 7: ELEMENTS OF A COMPUTER BASED DATA ACQUISITION SYSTEM [PAVLOU, Y. (1999)] 19
FIGURE 8: A PXI (PCI EXTENSIONS FOR INSTRUMENTATION) DATA ACQUISITION CARD 21
FIGURE 9: FIELDPOINT DISTRIBUTED DATA ACQUISITION SYSTEM 22
FIGURE 10: THE FP-AI-110 MODULE 24
FIGURE 11: A TERMINAL BASE WITH DUAL-CHANNEL MODULES 26
FIGURE 12 LOCATION OF SENSOR MODULES ON THE FLAGPOLE 28
FIGURE 13: DATASOCKET MODEL 36
FIGURE 14: THE ADVISE CYCLE 39
FIGURE 15: DATA MODEL FOR THE DATABASE 42
FIGURE 16: USER INTERFACE FOR A DATASOCKET SUBSCRIBER 50
FIGURE 17: LABWINDOWS/CVI DATASOCKET CLIENT 52
FIGURE 18: DATABASE ACCESS USING RMI 56
FIGURE 19: ACCESSING THE DATABASE THROUGH A SERVLET 63
FIGURE 20 IDENTIFICATION OF MODES OF VIBRATION 65
FIGURE 21 OFFSET ERRORS IN VELOCITY AND DISPLACEMENT 67
FIGURE 22 SCREENSHOT OF THE TUNED MASS DAMPER APPLET 68

LIST OF TABLES

TABLE 1: SALIENT CHARACTERISTICS OF THE CXL02LF LINE OF ACCELEROMETERS 18
TABLE 2: SALIENT CHARACTERISTICS OF THE FP-1600 MODULE 22
TABLE 3: SAMPLING RATES FOR FP-AI-110 24
TABLE 4: TRANSFER RATES FOR FP-1000 25
TABLE 5: SERVICES RUNNING ON THE DATA ACQUISITION SERVER 30
TABLE 6: DESCRIPTION OF DATASOCKET CONNECTION PARAMETERS 37
TABLE 7 TAGS FOR CVISERVER.CONF 40
TABLE 8: SERVICES RUNNING ON THE DATABASE SERVER 41
TABLE 9: CONFIGURATION FOR ARCHIVAL PROGRAM 49
TABLE 10: CONFIGURATION FOR THE RMI SERVER 61

CHAPTER 1. INTRODUCTION

1.1 MOTIVATION

We live in an environment where sensors are playing an increasingly important role in our daily lives. Revolutionary advances in MEMS (Micro-Electro-Mechanical Systems) have enabled the fabrication of sensors that are extremely small, consume very little power, and are yet highly accurate. Advances in sensing technology have been complemented by a corresponding acceleration in computing power, which is referred to as Moore's Law [Boriello, G., and Want, R., (2000)]. Major developments have also been made in signal processing and analysis of large data sets in real-time.

These advances in sensing, computation speed and signal processing, coupled with the rapid proliferation of networked devices and growth of wireless standards have made it feasible to create wireless sensor networks. These consist of clusters of collaborating "smart" sensors and enable extremely efficient monitoring, diagnostics and control [Kumar, S., et al., (2002) and Hung, E.S., and Zhao, F. (1999)].

Such networks of collaborating sensor systems hold a lot of promise for the monitoring and control of large civil engineering infrastructure. Indeed, they are central to the concept of an I-city, which envisages an entire metropolis linked to a web-based monitoring system. Such a system linked to appropriate diagnostic and control mechanisms could be invaluable during emergencies, helping to significantly reduce the loss of life and property.

On a smaller scale, MEMS sensors and actuators are already being used in the active control of motion-sensitive structures. These are structures having very stringent constraints on service load deflection, an ideal example being a silicon chip fabrication plant [Connor, J.J., (2001)]. The usual practice of deploying passive control devices like viscous dampers is gradually giving way to active (and semi-active) devices that provide a motive force to keep the deflection of the structure within the required limits. For an active control device to be effective, accurate and real-time measurements of the load and response of the structure are therefore important prerequisites.

It is strongly believed that efficient real-time monitoring coupled with fast simulation capabilities and control mechanisms can go a long way in ensuring safety and serviceability of Civil Engineering infrastructure.

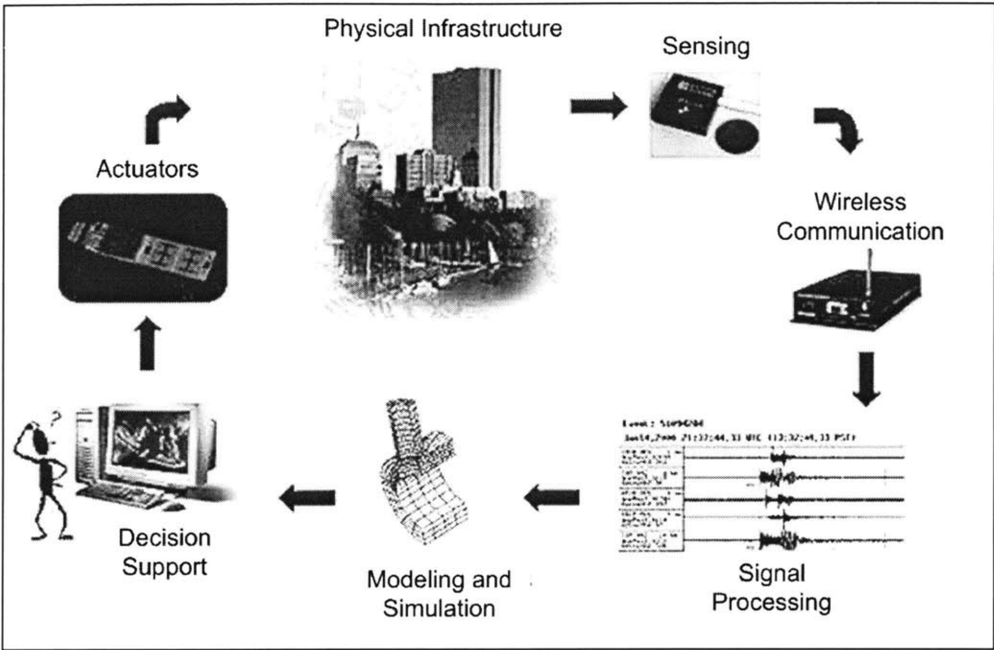


Figure 1: Illustration of the I-City Concept

The present research was motivated by the concept of an I-city, where sensing, signal processing, simulation and control come together to create smarter and safer physical infrastructure. A typical implementation of this concept is shown in Figure 1. Data from buildings and its surroundings is collected by sensors, and transmitted in real-time to a centralized cluster of servers using wireless transmission. The signals are then processed to remove noise and test for inconsistencies and fed to a fast simulation model. The results from the simulation are then processed to determine if corrective action needs to be taken. If so, actuators on the building are triggered to control its displacement, or, in the case of emergencies, the appropriate emergency personnel are immediately notified.

As an example, consider a building situated in a seismic region. The data collected from the sensors could consist of accelerations of the building and the foundation. The signal-processing processing could involve base-line correction of this acceleration data [Chopra, A.K., (1995)]. The simulation and modeling component might consist of integrating the acceleration to get the displacement at say, the top of the buildings. If these displacements

are excessive, as in the case of an earthquake, actuators throughout the building could be triggered to damp out the response.

If the seismic event were to be extremely destructive, the simulation might then predict the collapse of the building; in such an event, emergency crews could be alerted, and the building could be evacuated immediately.

1.2 OBJECTIVES

The main objective of the current research was to design and implement a scaleable, real-time virtual laboratory to monitor physical infrastructure. This would cover the monitoring, transmission and signal processing aspects mentioned in the previous section. The goal was also to experiment with the state-of-the-art in sensing and monitoring, including MEMS devices and emerging wireless standards like IEEE 802.11 and Bluetooth. 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 simulation tools.

The project also aimed at creating educational tools for enhancing the understanding of structural behavior and to serve as a platform for designing and testing active control algorithms. Some of these tools are also described in detail in subsequent chapters.

Unlike the more ambitious I-City, the current research was focused on developing a proof of concept example involving all the aspects discussed above. Therefore, a more tractable structure, a flagpole in DuPont courtyard on the MIT campus, was chosen. The location of the flagpole, which was measured to be 102 ft high, with a base diameter of 16 inches is shown in Figure 2.

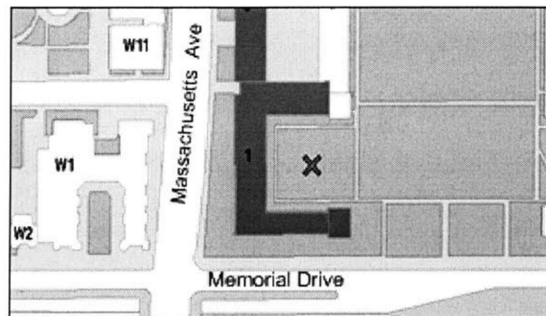


Figure 2: Location of the Flagpole in the DuPont Court (Marked With a Cross)

The main parameters monitored were accelerations along two orthogonal directions at three points along the length of the flagpole. A thermocouple was also added to monitor ambient temperature. Adequate care was taken during design and implementation to ensure that the developed framework could be applied with few modifications to larger, more complicated structures with a number of different sensors.

1.3 RELATION TO OTHER I-LAB PROJECTS

The flagpole remote instrumentation laboratory (http://flagpole.mit.edu) is one of several I-Labs under the I-Campus framework at MIT. Some of these laboratories include the microelectronics web lab (http://weblab.mit.edu), the photovoltaic weather station (http://pvbase.mit.edu/index.html), and the web accessible heat exchanger laboratory (http://heatex.mit.edu).

What differentiates the current effort from many of the other I-Labs is that the laboratory model in this case is a real-world structure in an outdoor environment over which no control can be exercised. Moreover, installation of the sensors itself was a very challenging task due to the height of the flagpole. Therefore, the fabrication and mounting of sensor packages had to be done very carefully to ensure reliable operation of the system over many years of outdoor use.

There were also many challenges in the implementation of the software component of this effort. For example, due to the high sampling rate for the accelerometers, the data acquisition server had to handle a large processing load and as a result, was not able to handle data archival efficiently. Therefore, a distributed solution was implemented with the database server and the data acquisition server functioning on two different machines. As expected, this caused many problems during the implementation of the server side software, many of which are discussed in this thesis. A few challenges encountered in the hardware and software aspects of this project are also discussed in [Greene, D.C., (2001)] and [Nelson, J.M., (2001)] respectively.

1.4 ROADMAP OF THE THESIS

This thesis is organized as follows. Chapter 2 discusses the hardware aspect of this research work. The exposition is kept to the level necessary for comprehending how the software interfaces with the hardware to sample data. Details on how the sensors work and how they are placed can be found in [Greene, D.C., (2001)]. Chapter 3 focuses on the software aspect of data collection and dissemination through the Internet. An overview of the development environment is given, followed by details on interfacing data acquisition software to the instrument. This is followed by details on how to broadcast the acquired data using TCP/IP sockets using the multithreaded DataSocket API [National Instruments, (1999a)]. The presence of two or more computers acquiring and processing data in tandem creates problems of reliability and uptime of the system as a whole. Therefore, some techniques for maintaining reliable operations between collaborating servers that were implemented are also mentioned. Chapter 4 deals with archiving real-time data in a database for subsequent processing. It mainly deals with the design of the database and the implementation of the Java interface to communicate with it. Chapter 5 deals with the processing and visualization of live and archived data. It discusses client side code for retrieving data from a DataSocket server by programs written in LabWindows/CVI and Java. It also discusses techniques to retrieve data from a database using the Java Remote Method Invocation API [Sun Microsystems, (1999)]. Chapter 6 then deals with how the framework implemented in the previous four chapters can be used to enhance the classroom experience in learning fundamental concepts in structural mechanics. Finally, Chapter 7 concludes the material presented in the thesis and outlines further work.

CHAPTER 2. SENSORS AND DATA ACQUISITION HARDWARE

2.1 OUTLINE

This chapter examines the hardware aspects of the remote monitoring project. The primary focus is on the sensors used for monitoring structural systems such as the flagpole. The section on data acquisition systems focuses on the distributed system, FieldPoint, manufactured by National Instruments and describes the various data acquisition modules and their characteristics. Detailed discussion of many of the hardware aspects can be found in [Greene, D.C., (2001)]. At the end of the chapter, the location and installation of the accelerometers on the flagpole are briefly described.

2.2 SENSORS

Roughly, a sensor converts a measurable physical quantity from one form to another that can be easily characterized and measured. For instance, an accelerometer converts acceleration 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 accelerometer can be used to measure its acceleration.

This section discusses two types of sensors that were used in the flagpole project, accelerometers (which measure acceleration, shock and vibration) and thermocouples (which measure temperature).

2.2.1 ACCELEROMETERS

An accelerometer can be formally defined as a transducer that converts mechanical motion into an electrical signal that is proportional to the acceleration vector along its sensitive axis [Crossbow Technology, (2000)]. The accelerations are in turn measured by measuring the force generated in a proof mass when in moves relative to its casing. A simple model of an accelerometer is that of a single-degree of freedom system, shown in Figure 3.

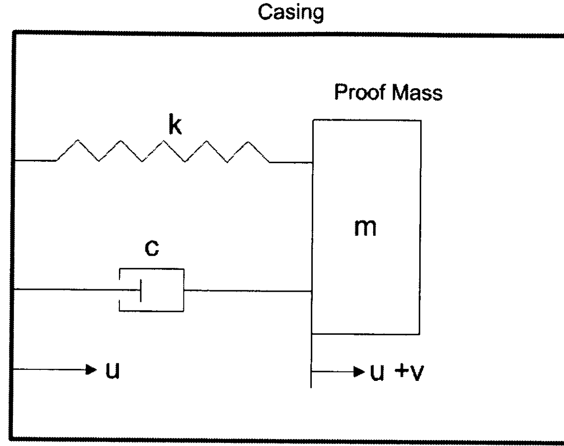


Figure 3: A Simplified Model of an Accelerometer

The equation of motion of the proof mass can be written as:

$$m\ddot{v}(t) + c\dot{v}(t) + kv(t) = -m\ddot{u}(t) \tag{2.1}$$

where, v is the relative displacement of the proof mass with respect to the casing and u is the absolute displacement of the accelerometer casing.

The force $f(t)$ exerted on the support (the casing) is:

$$f(t) = c\dot{v}(t) + kv(t) = -m(\ddot{u}(t) + \ddot{v}(t)) \tag{2.2}$$

Therefore, the acceleration of the casing, $\ddot{u}(t)$, can be estimated as:

$$\ddot{u}(t) = -\left(\frac{f(t) + m\ddot{v}(t)}{m}\right) \tag{2.3}$$

Therefore, the acceleration can be computed from Equation (2.3) by measuring the relative velocity, v and computing the force $f(t)$ according to Equation (2.2).

An accelerometer is often characterized by its resonant frequency, ω and damping ratio, ζ. These are defined as

$$\omega = \sqrt{\frac{k}{m}} \tag{2.4a}$$

$$\zeta = \frac{c}{2\sqrt{km}} \tag{2.4b}$$

Accelerometers can be classified on the manner in which the relative displacement, $v(t)$, and consequently, the force, $f(t)$, are measured. In *photoelectric* accelerometers, the proof mass deflects in between a photocell and a light source. The current obtained from the photocell then depends on the extent to which the mass obstructs the light source [Khazan, A.K., (1994)]. On the other hand, in the case of *piezoelectric* accelerometers, the mass exerts a force on a set of quartz crystals, and depending on this force, the charge produced by the crystals varies. In the case of *capacitative* accelerometers, the proof mass is constrained to move between the plates of a parallel plate capacitor. The net capacitance of the system is then determined by the relative location of the mass between the plates. In a more traditional version of this concept, the mass is constrained to move between the casing and one of the plates of a parallel plate capacitor. However, this approach tends to increase the size of the accelerometer [Khazan, A.K., (1994)].

Accelerometers are now-a-days fabricated using MEMS technology, either using *bulk* micro machining or *surface* micro machining of silicon or quartz wafers, which have near perfect crystalline structure and desirable mechanical properties. In bulk micro machining, a component is fabricated by etching it out of a wafer, as shown in Figure 4.

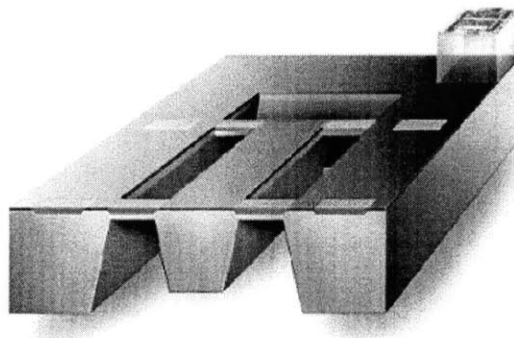


Figure 4: Illustration of Bulk Micro Machining ([Doscher, J., (1995)]).

Normally, this technique is not used for piezoelectric accelerometers, since, when fabricated in this manner, they tend to have non-linear characteristics and require very precise temperature compensation [Doscher, J., (1995)]. On the other hand, the bulk micro machining technique has been very successfully applied to capacitative accelerometers, which can be made very precise and sensitive. A schematic of such a capacitative accelerometer is shown in Figure 5.

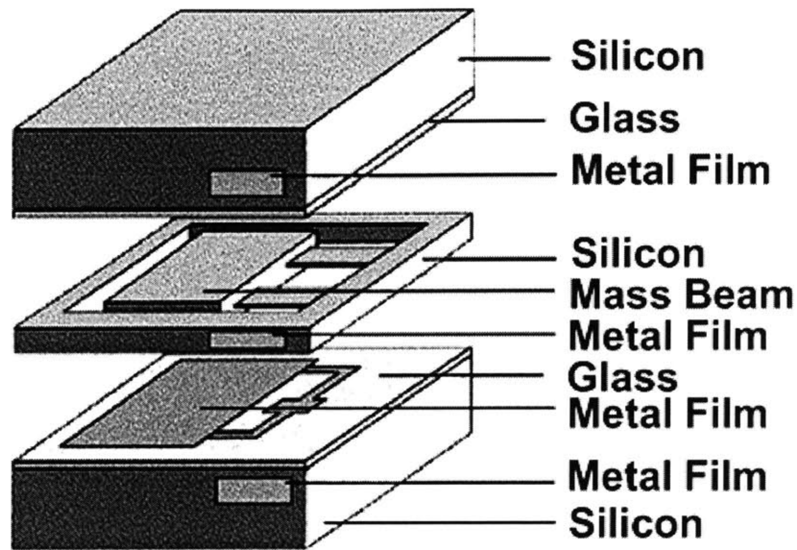


Figure 5: A Capacitative Bulk Micro Machined Accelerometer [Crossbow Technology, (2000)]

While being more accurate, bulk micro machined accelerometers tend to be relatively expensive.

Surface micro machined accelerometers are fabricated by depositing layers of poly silicon material on top of the silicon wafer, and then etching away parts of the material to form the sensors [Doscher, J., (1995)]. This fabrication technique is ideally suited for capacitative elements since each layer deposited can act as one of the plates of a parallel plate capacitor. Accelerometers fabricated in this manner are substantially smaller (they are usually packed as integrated circuits) and less expensive than those fabricated using bulk micro machining, though they are also less accurate. Such accelerometers are usually used to trigger air bags in cars, which are relatively low accuracy applications.

For the flagpole instrumentation project, bulk capacitative accelerometers manufactured by Crossbow were used. For testing the software, the CXL10LP1Z accelerometer mounted on an inverted cantilever prototype was used, which can measure accelerations along a single axis with a sensitivity of 500 milliVolt/g. For installation on the flagpole, three CXL02LF3 accelerometers were used, which are triaxial with a sensitivity of 1 V/g. Other salient properties these accelerometers are listed in Table 1:

Acceleration range	± 2g
Supply voltage	5 V
Zero acceleration voltage	2.5 ± 0.15 V

Table 1: Salient Characteristics of the CXL02LF Line of Accelerometers

The accelerations can be obtained from the voltage output by the following simple linear relation:

$$a = \frac{s * V - V_o}{\varepsilon} \tag{2.5}$$

Where, a is the acceleration, s is the scale factor for the measured voltage, V is the measured voltage, V_0 is the zero acceleration voltage output and ε is the sensitivity.

2.2.2 THERMOCOUPLES

Thermocouples are the most commonly used devices for measuring temperature. Nearly all types of thermocouples are based on the Seebeck effect, which is a term used to describe the voltage differential produced between two junctions at different temperatures, connected in a closed circuit by two different materials.

Thermocouples are classified depending on the materials used. The common varieties are Type J (Iron-Constantan), Type K (Chromel-Alumel), Type T (Copper-Constantan) and Type S (Platinum-Platinum10%Rhodium).

For the flagpole instrumentation project, a K-type thermocouple was used.

2.3 DATA ACQUISITION

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.

A typical computer based data acquisition system is shown in Figure 6.

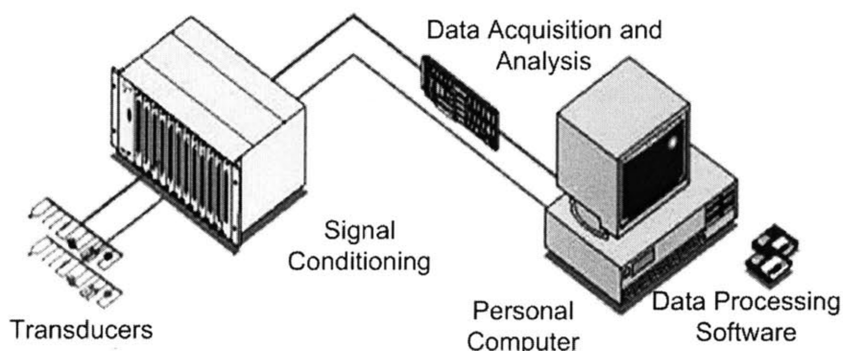


Figure 6: Elements of a Computer Based Data Acquisition System [Pavlou, Y. (1999)]

The (analog) signals from the sensors are first usually processed by a dedicated signal-conditioning unit. This unit pre-processes the signal before it reaches the data acquisition hardware. For instance, the signal conditioning unit might amplify weak signals or act as an isolation barrier between the computer and the sensors, blocking high voltage transients. The signal conditioning unit can also perform common filtering tasks like noise removal and anti-aliasing. Finally, the signal conditioner can power the sensors and provide bridge completion circuits.

The conditioned signal travels to the data acquisition hardware that converts the signal from analog to digital by sampling it at a predetermined rate, and feeds the samples into the computer. The effectiveness of the data acquisition hardware depends primarily on two factors: its *resolution* and *sampling rate*. The resolution determines the number of bits used to represent an analog signal. For instance, a device with a 12 bit resolution can chop up a signal into 4096 (2^{12}) levels, whereas a device with 16 bit resolution can chop up a signal into 65536 (2^{16}) levels. The sampling rate determines the rate at which the continuous analog signal is represented by discrete point values, or samples. The Nyquist theorem states that an Analog to Digital converter sampling at a frequency, *f*, cannot represent signals of frequency larger than $\frac{f}{2}$; the higher frequencies get *aliased* or folded into the lower ones. Many data acquisition devices have dedicated on-board processors, DSP chips and timers that enable them to handle tasks that would normally be handled by the CPU, significantly improving the performance of the system as a whole. Data acquisition devices plug directly into the host computer and communicate with it using architectures such as PCI, USB or RS 232. Some data acquisition cards can also use the PCMCIA architecture and are hence suitable for portable data collection. Digitized data either can be buffered on the card itself, written to the computer's memory using DMA (direct memory access), or transferred to the computer's hard disk. Figure 7 shows a PXI (PCI eXtensions for Instrumentation) data acquisition system manufactured by National Instruments.

Figure 7: A PXI (PCI eXtensions for Instrumentation) Data Acquisition Card

Finally, data acquisition software running on the computer periodically polls the card or the main memory to download the buffered data. This data can then be analyzed in different ways, displayed, transmitted over a local area network, or archived for later use.

One can observe that such a PC based data acquisition system, while having high performance (with up to 24 bit resolution, sampling rates in the kS/s and MS/s range and a variety of DSP options) can be extremely unwieldy and expensive. Moreover, such a system is not desirable in a distributed setting, where the data acquisition server needs to communicate with multiple data acquisition installations. Therefore, for the remote monitoring problem undertaken, the following characteristics were desired in the data acquisition hardware:

1. It must be capable of acquiring data on its own, buffering it, and transmitting the data to a central server when required
2. It must have integrated signal conditioning and data acquisition capabilities with sufficiently high resolution and sampling rates.
3. It must be easily upgradeable, and must accept a variety of sensor inputs.
4. It must be inexpensive to maintain, and be rugged enough for use in harsh environments.
5. It must use protocols that can be integrated with existing wireless standards.

Based on the requirements listed above, the FieldPoint distributed data acquisition system, manufactured by National Instruments was found to be suitable for the project. Apart from having many of the desirable characteristics mentioned, the hardware came a C library that

could be easily interfaced with the software development environment from National Instruments, making it very easy to write the data acquisition software.

The FieldPoint system consists essentially of one or more sensor input modules connected to a network interface module, as shown in Figure 8.

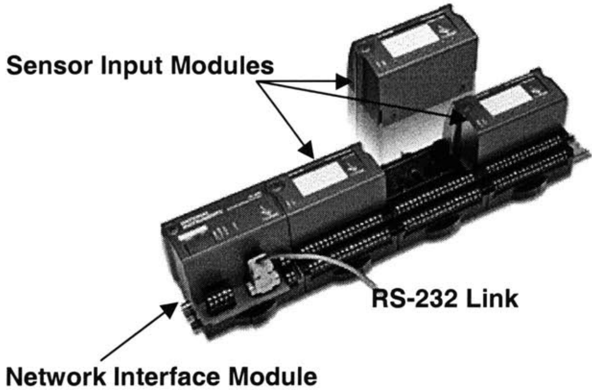


Figure 8: FieldPoint Distributed Data Acquisition System

The sensor input modules accept a wide variety of sensor outputs, including those from accelerometers, strain gauges and thermocouples, at different sampling rates and bit resolutions. Many of them also have basic signal processing capabilities (usually a user customizable low-pass filter) built in. The network module then communicates with the host computer using RS 232 or TCP/IP to transfer data. All FieldPoint units support plug-and-play customization of sensor input modules, and hence are modular and easily upgradeable. In the following paragraphs, some of the modules used in the monitoring project are discussed in detail.

2.3.1 ETHERNET NETWORK MODULES

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. Other salient characteristics of the module are given in Table 2:

Power Input	11 to 30 V DC
Power Consumption	7 W + 1.15 (Power for I/O Modules)
Operating Temperature	0 to 55 deg. C.

Table 2: Salient Characteristics of the FP-1600 Module

The FP-1600 module can be configured using the FieldPoint Explorer program. Configuring the device involves assigning an IP address and configuring the modules attached to it. *It must be ensured that the module and the computer used to configure it are on the same class B subnet (i.e. the first two fields of the IP address must be identical) 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 software such as LabView or Measurement Studio. Once the device has been configured as described, it can be assigned another IP address only by resetting it to factory defaults, as described in the FP-1600 User Manual [National Instruments, (1999b)].

One of the main drawbacks in the FP-1600 module is the lack of an on-board memory buffer. Due to this limitation, each sample from the sensor modules is sent through the network in its own TCP/IP packet, severely overloading the network. This also has serious repercussions on the quality of rapidly sampled data, like the ones from the accelerometer, due to dropped packets, especially in an IEEE 802.11b wireless network.

The next generation of Ethernet modules fall into the FP-20xx category. These come with on-board memory buffers, 3 MB in FP-2000 and 11 MB in FP-2010. In addition, they also support a subset of the LabView RT operating system, which enables them to run real-time embedded processes written in LabView. These modules can communicate in two modes, the traditional publisher-subscriber mode and in a peer-to-peer mode. This enables different FieldPoint modules to share data and instructions between themselves and the host computer enabling truly distributed data acquisition and processing.

2.3.2 SERIAL NETWORK MODULES

The FP-1000 modules communicate with a host computer using the RS 232 protocol. These have data transfer rates with the host computer of up to 115.2 kb/s, and can support up to nine sensor input modules.

The modules can be configured with the FieldPoint Explorer program described earlier. The FP-1000 module must be directly connected to the serial port of the host computer and the RS-232 interface must be chosen. The communication speed specified on the module must be then selected. The program then configures all the sensor input modules on the unit, which can again be saved as an IAK file.

2.3.3 ANALOG INPUT MODULES

The FP-AI-110 modules support up to eight channels of voltage or current inputs with 16 bit resolution. They also come with user programmable low-pass filters at 50,60 and 500 Hz settings, which can be used to filter out high frequency noise from the data. The FP-AI-110 module is shown in Figure 9.

Figure 9: The FP-AI-110 Module

According to the National Instruments sampling benchmarks [National Instruments, (1998)], the speed at which a host computer can acquire data from a FieldPoint installation depends upon two independent factors, the *sampling rate* of the sensor input module and the *network throughput rate*. The *sampling rate* of the module (also referred to as the update rate) is defined as the rate at which the Analog to Digital converter in the module digitizes the input and places it in the output register [National Instruments, (1999c)]. 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 Table 3.

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

Table 3: Sampling Rates for FP-AI-110

The *network throughput rate* is the rate at which the network interface module transfers data between the FieldPoint installation 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. Due to the high-speed bus linking the network module to the sensor input modules, the time taken for the network

module to read data from the sensor input modules is negligible compared to these two rates. Table 4 shows a few typical transfer rates for the FP-1000 module connected to one analog input module, such as FP-AI-110 [National Instruments, (1998)].

	BAUD RATE				
	115.2 kb/s	**57.6 kb/s**	**38.4 kb/s**	**19.2 kb/s**	**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 4: Transfer Rates for FP-1000

The overall sampling rate is determined by which of the two rates actually governs. For example, consider an eight channel FP-AI-110 module connected to a FP-1000 module. The sampling rate for a 500 Hz filter is 0.17 s or 170 ms (see Table 3), whereas the network throughput rate at 57.6 kb/s is 17 ms (Table 4). Therefore, the module sampling rate determines the Nyquist frequency. Since the governing sampling rate in this case is 5.88 Hz ($\frac{1}{0.17}$), the Nyquist frequency is 2.94 Hz ($\frac{1}{0.34}$).

On the other hand, if one uses a high-speed module such as FP-AI-100, which can sample at 2.8 ms (but which comes only with one fixed low-pass filter at 120 Hz and 12 bit resolution), the actual sampling rate is governed by the network throughput instead of the sampling rate of the analog to digital converter.

In the present monitoring project, if was found that the sampling rate of the FP-AI-110 module was very unsatisfactory for sampling accelerometer data, where rates of up to 100 samples/sec were desired. Hence, dual channel modules were used, which are described next.

2.3.4 DUAL CHANNEL MODULES

The dual channel modules have two channels of input and instead of transmitting data directly to the network module, plug into a terminal base (FP-TB-10). Each FP-TB-10 terminal base can hold up to six dual channel modules, each of which can accept a different kind of sensor. They are therefore more versatile than the analog input modules, each of which accepts only one type of sensor. Moreover, by utilizing only the modules that are needed, it is possible to increase the overall performance of the system.

A terminal base with modules is shown in Figure 10.

Figure 10: A Terminal Base with Dual-Channel Modules

The dual-channel modules were used to acquire acceleration data. The specific module used was FP-AI-V10, which has a voltage input range of 0 to 10 V and a 12 bit resolution. More importantly, it has a sampling rate of 2.8 ms, which was found to be sufficient for sampling accelerations.

2.3.5 THERMOCOUPLE MODULES

The FP-TC-120 thermocouple modules can take up to eight thermocouple inputs and provide a 16 bit resolution. They can be configured at different temperature ranges and with different types of thermocouples and conveniently provide a direct temperature reading. The sampling rate of these modules was not a limiting issue because the samples were read only once very 160 ms, which was easily handled.

2.4 WIRELESS SENSOR NETWORKS

As mentioned earlier, one of the aims of this research was to explore existing technology in wireless sensor networking for large civil engineering infrastructure. There are essentially two approaches to this.

The first approach is to create a wireless link between the host computer and multiple data acquisition devices, while physically connecting each sensor to a data acquisition device in its proximity. This approach requires the data acquisition device to have its own networking and processing capabilities. Therefore, it is more appropriate for a FieldPoint installation than it is for a PXI card based data acquisition system.

The second, more fundamental approach is to make the connection between the sensors and the data acquisition device wireless. This approach is the heart of the wireless integrated sensor network (WINS) concept [Pottie, G.J., and Kaiser, W.J., (2000)], which is a generalization of the I-City concept, with applications including medical informatics and defense. While this is being promoted as a low power alternative to the first approach, several issues remain to be addressed, such as signal attenuation between the sensor and the data acquisition system and the sampling rates attainable. In this section, the first approach will be discussed since it was implemented in the virtual laboratory.

2.4.1 WIRELESS LINK BETWEEN HOST AND DAQ SYSTEM

For this approach, two alternatives were considered. The first alternative recommended by National Instruments consists of a pair of radio modems in a master-slave configuration. While the master modem connects to the host computer, the slave connects to a FieldPoint network interface module (FP-1000, FP-2000 or FP-2010). The radio modems use the 902-928 MHz RF band and provide up to 114 kb/s transfer rate with a line of sight transmission of over 20 miles. Within buildings however, no more than a few hundred meters are promised. The radio modems can also function as repeater to extend this range. Unfortunately, these devices are extremely expensive (costing around $3,600 for a master-slave pair).

The alternative to radio modems was to use off-the-shelf wireless networking kits manufactured by Lucent technologies. These devices use the IEEE 802.11b (also called WiFi) protocol, which enables data transfer rates of up to 11 Mb/s, and uses the 2.4-2.485 GHz spectrum for communication. The wireless network topology chosen for the project consists of one or more FieldPoint installations and a host computer connected to individual wireless network cards and communicating with a centralized router, known as the residential gateway. While the range of the wireless cards is variable (the card automatically reduces the data transmission rate to increase range), the residential gateway provides up to 150 m of roaming access in open areas. This is obviously reduced within enclosed spaces. A third component necessary for building a wireless network infrastructure is the Ethernet converter, that takes serial or Ethernet inputs and connects to a wireless network card.

While this technology was found to be quite feasible for projects with low sampling rates, it was found that for sampling rates of the order of a 100 Hz, the Ethernet module, FP-1600

resulted in very poor data throughput. This was because of the lack of a data buffer on the module, due to which an acquired sample was overwritten by a new sample before it could be transmitted. Therefore, as of now, the FieldPoint installation is directly connected to the host computer via a RS-232 link.

2.5 INSTALLATION OF SENSORS ON THE FLAGPOLE

The sensors consisting of three triaxial accelerometers protected by waterproof metal cases were mounted along the length of the flagpole according to the diagram shown in Figure 11 (Adapted from the installation schedule prepared by Matthew Echard, http://flagpole.mit.edu/Matthew/Sensor_Installation.pdf). The sensors were placed in order to be able to capture the positions of maximum deflection of the first three modes, the location of which were determined from finite element simulations

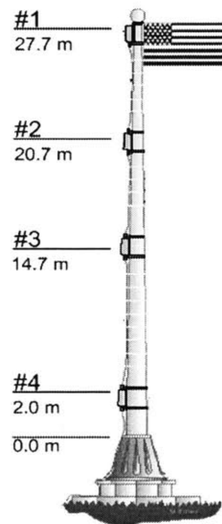


Figure 11 Location of sensor modules on the flagpole

The sensors alone were installed and tested on February 15, 2002. Later, an underground cable was laid between the flagpole and the data acquisition server. Since the system was adequately tested before deployment, no major changes were required in the hardware or software. Finally, after deployment, the accelerometers were recalibrated and the new zero acceleration voltage was changed in the data acquisition software. Java applets accessing real-time data from the flagpole are available at http://flagpole.mit.edu/realtime.html. The next few chapters describe the implementation of the server and client side software for the virtual laboratory, and describe in detail how data is collected and distributed across the network and accessed by Java clients.

CHAPTER 3. DATA COLLECTION AND DISSEMINATION

3.1 SOFTWARE FOR DATA ACQUISITION

This chapter discusses the software components necessary for a distributed data acquisition and processing system like the one considered in this research. The next chapter considers the issue of archival of real-time data and retrieval of this archived data.

As pointed out earlier, the FieldPoint sensor input modules sample the data from the sensors and communicate it to the network interface module of the installation. It is now up to the host computer to access and process this data by polling the instrument either through Ethernet or through a serial connection. While this would normally being a daunting task requiring low-level programming, National Instruments simplifies this task by providing two software solutions.

The more popular product, LabView (http://www.ni.com/labview), uses a graphical programming language, where block diagrams or VIs (Virtual Instruments) are used for programming tasks such as interfacing with the instrument, publishing data to a network, visualizing the acquired data etc. However, the functionality is restricted to what such a graphical programming environment can provide. It is difficult for instance, to make native Win32 API calls, spawn off external processes, etc.

An alternative to LabView is Measurement Studio (http://www.ni.com/mstudio), which consists of three components, LabWindows/CVI, which is a ANSI C compliant programming interface, Component Works, which is a Microsoft Visual Basic interface and Component Works++, which is an add-in to Microsoft Visual C++. It was decided to use LabWindows CVI to develop the data acquisition software because of its convenient interface to the FieldPoint network modules. There was also provision to spawn off external Java programs that were necessary for data retrieval. The next section discusses the data acquisition software in detail and explains its working.

As discussed in the previous chapter, data from the three accelerometers is polled once every 10 milliseconds and thermocouple readings are sampled once every 160 milliseconds. It was found that making the same computer handle the database archival and retrieval placed additional load on the system due to constant read/write operations, which interfered with its ability to effectively read data from the instrument. Therefore, two machines were used, one

of them was used for acquiring data from the instrument and the other was used for archiving data as well as for serving static web pages. An additional complication arose during the development of client-side Java applets to read real-time and archived data. The Java virtual machine allows an applet to make network connections only to the host from which it originated. Therefore, an applet residing in the data acquisition server accessing real-time data could not at the same time, access archived data. Similarly, an applet residing in the database server could access archived data but not real-time data. Therefore, the Java Remote Method Invocation API [Sun Microsystems, (1999)] was used to enable applets hosted on the data acquisition server to make queries to the database by calling methods on a remote object exported by the database server. This allowed an applet to access both real-time as well as archived data. The full implementation of this RMI framework is discussed in the next chapter. The table below lists the services running on the data acquisition server.

SERVICE	PURPOSE
`menu.exe`	Collects data from the FieldPoint installation and publishes it to a DataSocket server
`cwdss.exe`	National Instruments DataSocket server
`rmiserver.class`	Class implementing the RMI interface which allows applets to access the remote database by making SQL calls on a remote object.

Table 5: Services running on the data acquisition server

3.2 INSTRUMENT INTERFACE WITH LABWINDOWS/CVI

Acquiring data from a FieldPoint instrument involves six essential steps. These are discussed in detail below. The entire program can be downloaded from the CVS repository at http://wavelets/cgi-bin/cvsweb.cgi/cviserver/?cvsroot=Flagpole.

3.2.1 STEP 1: CONFIGURING INSTRUMENT AND MODULES

First, the FieldPoint installation is configured using the FieldPoint Explorer program as described in the previous chapter. The unused channels in all instruments must be removed to minimize data transfers. The instrument configuration must then be saved as an IAK file.

3.2.2 STEP 2: LOADING INSTRUMENT DRIVERS INTO CVI

Next, all the necessary external instrument libraries must be loaded into the CVI environment. These libraries are similar to DLLs and are handled by the compiler in a similar fashion.

3.2.3 STEP 3: CONNECTING TO THE INSTRUMENT

The third step involves getting a connection handle to the FieldPoint installation. This is done as follows:

```
133:        /* Open a connection with the instrument using the last saved .IAK file*/
134:        if (status = FP_Open (NULL, &FP_handle)) {
135:              Error(status);
136:        }
```

The NULL option in the function indicates that the last saved IAK file must be used for configuring the instrument. Instead of this option, any IAK file might be specified. If connection to the instrument is successfully established a valid handle is returned.

3.2.4 STEP 4: GETTING A HANDLE TO INSTRUMENT CHANNELS

The next step is to create a reference to each channel that is to be monitored. The following piece of code demonstrates how this can be done for the thermocouple module:

```
146:        /* code for the thermocouple */
147:        if (status = FP_CreateTagIOPoint(FP_handle,"FP Res",module[1],"Channel 0",
                                              &IO_handle[numChannels]))
148:                Error(status);
```

Here, the first argument is the handle returned by the `FP_Open` statement, the second argument is the sensor input module name (`module[1]`), the third argument is the channel to be monitored, and the fourth argument is the handle to the channel passed by reference.

3.2.5 STEP 5: CREATING AN ADVISE OPERATION ON THE CHANNELS

Once a valid handle/pointer has been returned to each of the channels to be monitored, the next step is to actually start polling the instrument to get the data into the main memory. This is termed as an *advise* operation. The following important parameters must be specified for the advice cycle. First, the handle to the instrument (obtained from `FP_Open`) must be passed. Second, the handle to each channel to be monitored (obtained from `FP_CreateTagIOPoint`) must be supplied. Third, the advise rate, which is the rate at which the instrument is polled for values must be provided. Forth, a memory buffer must be specified to hold the data from the channel. This buffer must be a global array so that it retains scope when the data is harvested elsewhere within the program. Finally, an optional callback function may be provided that is be triggered whenever data becomes available in the memory buffer. Such a callback might be used to get the data off the temporary buffer and process it further, for instance, send it across the network. Callback functions can be configured to be *notify-on-change*, which means that the function is triggered only if the new data in the buffer at a particular polling instance is different from the data already present in the buffer. This feature is useful for monitoring slow events. Callback functions can also configured to be *asynchronous* or *post-deferred.* In asynchronous callbacks, each callback operation runs in its own thread, and is hence independent of the main thread. On the other hand, for post-deferred callbacks, the callback runs in the same thread as the main program, and hence restrictions on the size of the functions and the operations performed by it are quite severe and must be scrupulously followed.

Despite making provisions for callback functions, National Instruments discourages their use due to performance reasons. The recommended method for processing buffered data is to create a timer instance that then calls its callback function every time it "ticks". This callback function can then be used to read data off the memory cache and process it further. The timer recommended for this purpose (and used in all demonstration programs) is the general purpose UI timer that ships with LabWindows/CVI. However, it was found that for very high advise rates, the UI timer proved to be very CPU intensive. Therefore, an asynchronous timer object was used that is implemented by making low level API calls to the operating system and was found to be satisfactory even for sampling rates of up to 100 Hz. A caveat that comes with using this timer is that it is not very reliable for time intervals less than 10 ms, which is a limitation of the operating system itself. Therefore, it was noticed for a sampling interval of 10 ms, while the time interval between each sample was not always exactly 10ms, on an average, each minute had 6000 samples of data. This implies that there is underlying drift correction code built into the event triggering framework.

The following code illustrates the code necessary to trigger an advise operation on the Thermocouple:

```
149:        if (status = FP_Advise(FP_handle,IO_handle[numChannels],10,0,
                  advisebuf[numChannels],100,1,NULL,NULL,&advise_ID[numChannels]))
150:        Error(status);
```

In the function call shown above, the first and second parameters are the handles to the FieldPoint installation and the channel respectively, the third parameter is the advise rate in milliseconds. The fourth parameter is a notify-on-change flag (which in this case does not play any part), the fifth parameter is the array buffer for caching the data, the sixth parameter is the buffer size in bytes, the seventh parameter is the flag for the type of callback (asynchronous or post-deferred, also not important in this case). The eighth parameter is the function to be used as callback (this is `NULL` and is ignored), the ninth parameter is the event data to be passed to the callback (also ignored). Finally, a handle is passed by reference to the `FP_Advise` function, which is needed later on for retrieving data from the buffer.

After initiating the advise operation, the asynchronous timer must be then instantiated and started. This is done in the following segment:

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

Notice that a callback function, `adviseCB` is passed. This is triggered after the specified interval, which is specified as the first argument to the function.

3.2.6 STEP 6: READING AND PROCESSING CACHED DATA

The final step is to read the data cached in the buffer, `advisebuf`, at the end of each advise cycle (Recall that each advise operation has its own buffer). This is done in the timer callback function, `adviseCB`, using the `FP_ReadCache` function.

This is illustrated for the thermocouple in the following code segment:

```
184:            if(!DEBUG){
185:                    is = FP_ReadCache(FP_handle,advise_ID[numChannels],
                                    current_read,BUFFER_SIZE,&dummy);
186:                    value = (float*)(current_read);
187:                    channels[numChannels][2] = (double)(*value);
188:            }else
189:                    channels[numChannels][2] = 70+rand()/(32767.0);
```

In the call to the `FP_ReadCache` function, handles to the instrument and advise operation are passed. Also supplied are: a pointer, `current_read`, which points to the cached data, `BUFFER_SIZE`, the size of the cache and `dummy` and a time stamp structure (by reference).

The FP-1000 module has a bug due to which the returned time stamp only has a one second resolution. Instead, a system timer is used for timing the data. Once the pointer to the data is obtained, it is cast as a pointer to float (`float*`), and dereferenced to get the final value.

The steps illustrated above provide only the skeleton for the complete data acquisition server program. The entire code can be downloaded from the CVS repository (http://wavelets.mit.edu/cgi-bin/cvsweb.cgi/cviserver/src/cvi/?cvsroot=Flagpole).

Once data from the memory cache has been harvested, it is a relatively simple matter to process it in different ways. For instance, National Instruments has an extensive signal processing library in LabWindows/CVI to handle tasks like computing the spectrum, designing filters, etc. In the next section, it is shown how this data can be made available in real-time to clients on the Internet using the DataSocket APIs.

3.3 DATA DISSEMINATION USING DATASOCKETS

DataSockets is a collection of libraries targeted at many platforms specifically meant for sharing real-time data, which uses a publisher-subscriber model instead of a client-server one.

In a conventional client-server model a server, which has access to live data, listens for client connections on a TCP/IP port. When a client connects at that port, the server usually spawns a thread, which then handles subsequent communication between the client and the server on a different port. The main thread in the server however, continues to listen for further incoming client requests on the same port. Once the communication between the client and server is finished, the thread terminates, and before doing so, performs a number of clean-up operations. This model is quite straightforward to implement for protocols such as SMTP and HTTP. However, this is not very convenient for real-time data for a number of reasons. For one, the server is responsible for spawning off threads, handling communications and cleaning up after a particular transaction, which are quite tedious implementation details and can be very expensive to handle in a real-time scenario. In addition, when data types other than characters or integers (for instance, arrays or strings) have to be passed, the server and client have to decide on the protocol for marshalling and demarshalling the data at their respective ends. This is also a very tedious programming detail.

Instead of looking at the source of data as a *server*, and the application accessing it through the network as a *client*, the DataSocket API has the notion of a *publisher* and a *subscriber*, respectively [National Instruments, (1999a)]. In addition, there is a DataSocket server, which handles all the communication between the two. The publisher binds to a particular URI (with a prefix `dstp://`) on the DataSocket server and sends data to it. The subscriber then subscribes to data on the DataSocket server with the same

URI as the publisher, enabling them to share information. The server and the API take care of forking multiple connections, serializing the data into a stream of bytes on the subscriber side and deserializing it on the client side. The API also supports a simplified form of reflection, enabling the subscriber to determine the data type sent by the client dynamically. Figure 12 shows working of this publisher-subscriber model. As of now, DataSocket implementations exist for LabView, LabWindows/CVI, ComponentWorks, ComponentWorks++ and Java.

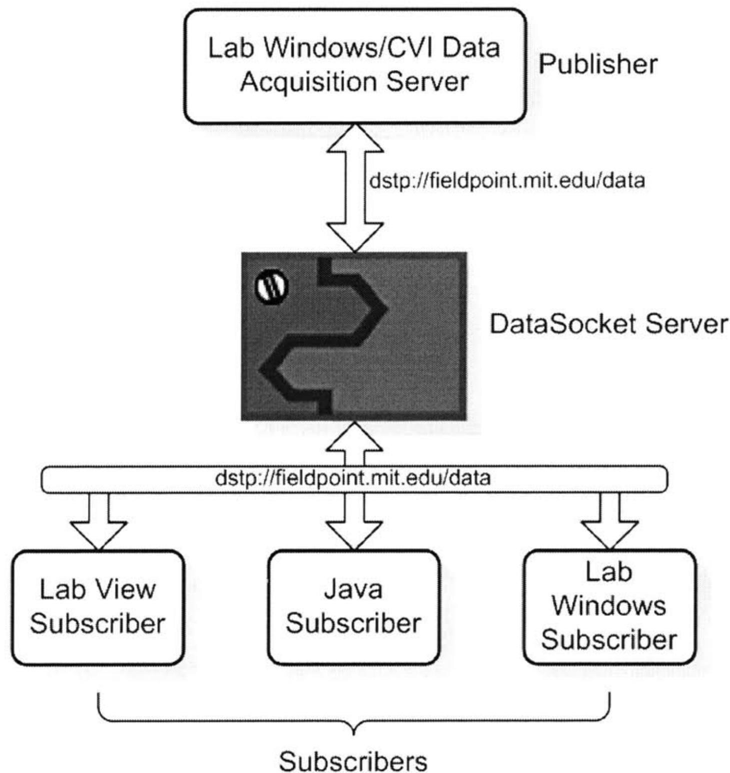


Figure 12: DataSocket Model

The next few items in this section describe how data can be written to a DataSocket server from a LabWindows/CVI program. It also describes the data model used in the flagpole monitoring project to optimize network transfers. Publishing the data to a DataSocket server involves two steps, getting the connection, and writing the data. This is described in detail next.

3.3.1 STEP 1: GETTING A HANDLE TO A DATASOCKET URI

To get a reference to a DataSocket URI to which data can be posted, the DS_Open function in the LabWindows/CVI DataSocket library can be used as shown in the following code segment:

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

The first statement gets the URI (line 77) to bind to, and the second statement actually binds to it. There are several modes in which a publisher or subscriber can bind to the DataSocket server. These modes and their properties are listed in the following table:

CONNECTION TYPE	PROPERTIES
DS_Read	Opens a connection as a subscriber. New data must be manually read using the DS_Update function. Callback function is ignored.
DS_ReadAutoUpdate	Opens a connection as a subscriber. Callback is triggered each time new data becomes available, or when the connection status changes.
DS_Write	Opens a connection as a publisher. Data must be written to the DataSocket server and manually updated using DS_Update. Callback is trigged when status changes.
DS_WriteAutoUpdate	Opens a connection as a publisher. Data is automatically updated to the server every time it is written to it. Callback is triggered with status changes.

Table 6: Description of DataSocket Connection Parameters

In the code illustrated previously, an automatic update is preferred, and hence the DS_WriteAutoUpdate connection mode is selected. The next parameter is the callback for DataSocket events. For a publisher instance, this callback may be NULL, but here it is provided because a DataSocket callback is triggered whenever the server status changes (for instance, if the connection with the server is broken). The parameter after

that is the data to be passed to the callback function. Here it is not needed and is hence ignored. The final parameter is a handle to the DataSocket connection, passed by reference. If a connection is successfully established, a non-`NULL` value is returned for the handle. Once a connection is established, data may then be written to the DataSocket server as described in the next step.

3.3.2 STEP 2: WRITING DATA TO DATASOCKET SERVER

The write operation is handled in the callback to the asynchronous timer (`adviseCB`).

It can be easily seen that publishing the data at the end of each advise operation (which occurs 100 times a second) can severely overload the processor and the network. In addition, an overhead is incurred in packing the data and its associated time-stamp. Instead, the program publishes the data in cycles, where each cycle consists of 16 advise operations on the dual channel modules and one advise operation on the thermocouple module. Time stamps are obtained only at the beginning and end of each cycle, and are assumed to be linearly varying across the data samples. The entire data block is packed in a 2-D array of `doubles` with `numChannels+1` rows and 16 columns, where `numChannels` is the number of accelerometer channels (6). The last row consists of three entries: the start time of the block (number of milliseconds since the Java epoch), the end time of the block (number of milliseconds since the Java epoch) and the temperature in Fahrenheit.

The time-stamp is computed in a slightly roundabout manner, since Windows and Java measure times in different ways. Windows has two functions for measuring time, `time`, which returns the number of seconds since 1st January 1900, 00:00:00 GMT, and `GetLocalTime`, which returns a `SYSTEMTIME` structure, with fields for the year, month, day of the month, day of the week, hour, minute, second and millisecond. Java on the other hand, uses as its epoch, the number of milliseconds since 1st January 1970, 00:00:00 GMT. Hence, at the beginning and end of each cycle, the Windows time-stamp is converted to a Java time-stamp so that it can be easily processed by Java clients. This is accomplished by the following piece of code:

```
161:        GetLocalTime(&dummy);
162:        localTimeInSeconds = time(NULL);
163:        localTimeInMillis = (localTimeInSeconds-secondsDiff)*1000.0+dummy.wMilliseconds;
```

Here, `secondsDiff` is the number of seconds between the Windows and the Java epochs. Once the data to be sent is packed into a 2-D array, it is written to a DataSocket server as shown in the next code segment.

```
191:                if(dsHandle){
192:                        hr = DS_SetDataValue (dsHandle, CAVT_DOUBLE|CAVT_ARRAY,
                                                channels,numChannels+1,16);
193:                }
```

Here, `dsHandle` is the reference to the DataSocket connection (obtained from `DS_Open`), `CAVT_DOUBLE|CAVT_ARRAY` denotes that the object being written to the server is an array of `doubles`, `channels` is the two dimensional array being written, `numChannels+1` is the number of rows, and the final parameter, `16`, is the number of columns.

Figure 13 summarizes the operations discussed in sections 3.2 and 3.3.

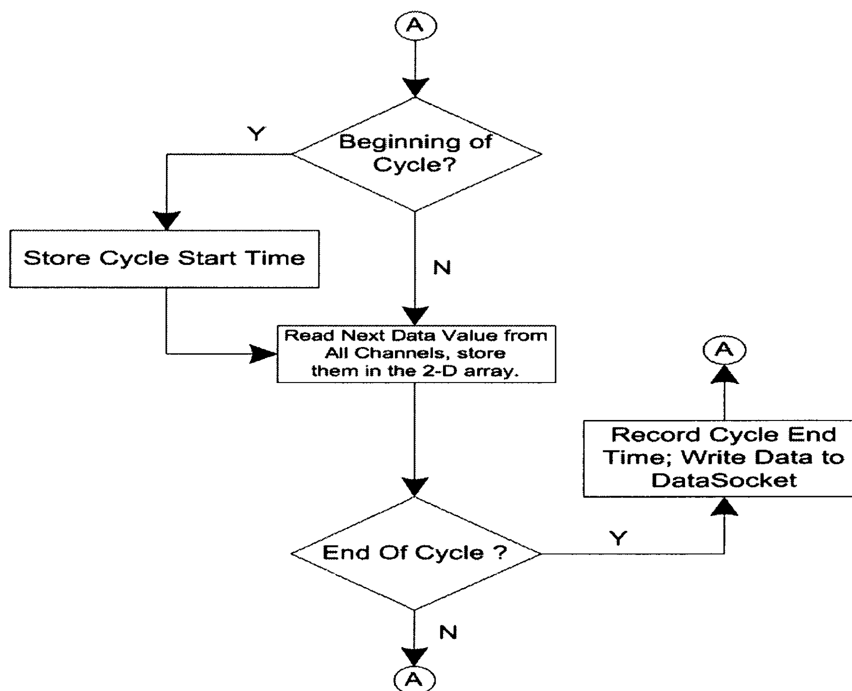


Figure 13: The Advise Cycle

This concludes the discussion of the publisher end of the DataSocket API. The subscriber end is discussed in chapters 4 and 5.

3.4 CONFIGURING THE DATA ACQUISITION SERVER

The data acquisition server has a number of options that can be configured using two XML-like configuration files, `cviserver.conf` and `rmiserver.conf`. The options for the first configuration file are discussed below; the discussion of the RMI framework is postponed until chapter 5.

TAG	PROPERTY
`<CHANNELS>`	Number of accelerometer channels
`<CALIBRATION>`	Calibration data for accelerometer
`<DEBUG>`	Run server in debug mode
`<AUTOSTART>`	Start all server operations automatically

Table 7 Tags for `cviserver.conf`

The first tag must be `<CHANNELS>` and must contain the number of accelerometer channels in the installation to be monitored. Then, the `<CALIBRATION>` tag for each accelerometer consists of three white-space delimited values: the scale factor for the voltage, the zero acceleration output, and the sensitivity. The acceleration is then computed using equation (2.5). It must be noted that the tags are case sensitive and must appear in the order listed. A sample configuration file is shown in the following listing.

```
1:    <CHANNELS>
2:    4
3:    <CALIBRATION>
4:    1.0 2.6007476 0.987
5:    <CALIBRATION>
6:    1.0 2.6031892 0.983
7:    <CALIBRATION>
8:    1.0 2.5494773 0.979
9:    <CALIBRATION>
10:   1.0 2.5836575 0.987
11:   <AUTOSTART>
```

CHAPTER 4. DATA ARCHIVAL

4.1 SOFTWARE FOR DATA ARCHIVAL

The previous chapter discussed the various software components for collecting data from sensors and publishing it on the network. This chapter describes the software framework for data archival, which is necessary for subsequent analysis and processing of older records. Details about many design iterations followed before converging on the final solution can be found in [Nelson, J.M., (2001)]

As mentioned earlier, the data archiving program runs on the web server, which is different from the data acquisition server. The following table lists the services running on this machine.

SERVICE	PURPOSE
`Archive.class`	Main data archival program
`JarReader.class`	Class implementing a remote method to extract data from Jar files and insert them into the database

Table 8: Services running on the database server

The next section discusses the design of the database. It includes a description of the tables and stored procedures.

The section after that presents the archive program and discusses its operation. Then, it is explained how the services running on the database server can be made "aware" of the services running on the data acquisition server, which is useful for ensuring reliable network operations. Finally, customization of the archival program is briefly discussed.

4.2 THE FLAGPOLE DATA MODEL

Data from accelerometers is first written to text files in a comma-delimited format, a new file being created each minute. Each text file is named according the minute to which it corresponds. At the end of each minute, the text file is uploaded into a Microsoft SQL Server 7.0 database and added to an hourly zip-file archive. The text file is then deleted to conserve disk space.

Data from thermocouples is directly inserted into the database, once every 16 seconds.

It was mentioned that the accelerometer data is sampled at 100 samples/sec. This implies that the number of samples in a day is well over 8 million. Keeping all this information in the database can be very expensive. Therefore, at the end of each hour, data from the previous day is purged from the database. Whenever data older than 24 hours is required for a database query, it is extracted on the fly from the zip archive and uploaded into the database by the `JarReader` service (see Table 8).

The database consists of three tables and six stored procedures. Figure 14 illustrates the data model for the database.

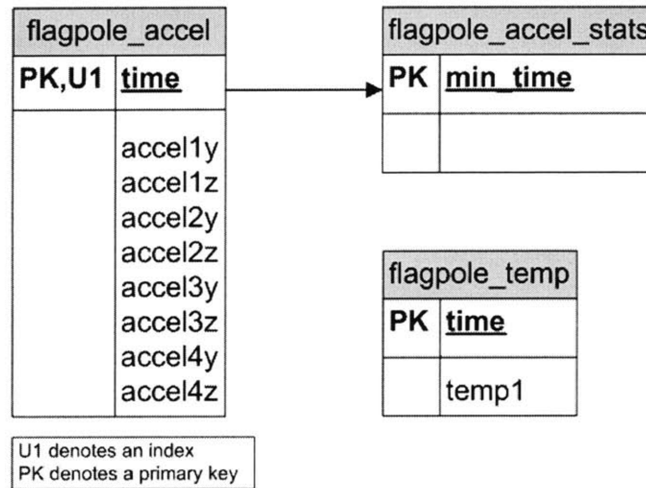


Figure 14: Data model for the database

The `flagpole_accel` table stores the acceleration data from four sensors along two axes as SQL `float` along with the time stamp as SQL `decimal`. The `flagpole_temp` table stores data from the thermocouple as SQL `float` and an associated time stamp as SQL decimal. The `flagpole_accel_stats` table holds the timestamp for the earliest record in the database. Therefore, when a query is made on `flagpole_accel`, it is first checked to see if the record exists in the database by reading the `min_time` field from `flagpole_accel_stats`. If the queried time is larger than `min_time`, the query is made immediately. If not, the `JarReader` service is called to extract the appropriate zip file and upload the data on to the database. The query is then executed against the uploaded entries. At the end of the hour, data uploaded from zip files, being older than a day, is automatically purged.

4.3 DATABASE ACCESS USING JDBC

The JDBC API [Hamilton, G., and Cattel, R., (1998)] provides Java programs access to tabular data sources in a large number of formats, including ODBC, spreadsheets or text files. While the core API provides a high-level, platform independent interface, it relies on platform and data format specific implementations to actually access and manipulate the data.

For accessing the database, the JDBC-ODBC driver, which ships with the JDK was used. While Sun recommends this driver be used only for testing (because of its use of native code and client-side configuration requirements), other drivers for Microsoft SQL Server 7.0 like AveConnect (http://www.atinav.com) and FreeTDS (http://www.freetds.org) lacked many features (like bulk insert) or performed poorly.

The centerpiece of the JDBC API is the `Connection` class. This is then used to generate an instance of the `Statement` or `PreparedStatement` interfaces. Once a SQL statement has been initialized, it is quite straightforward to execute queries and updates on the database. Details on obtaining connections and statements are illustrated in detail in a number of sources [Hamilton, G., and Cattel, R., (1998) and White, S., et al., (1999)].

The simplest scenario described in [Hamilton, G., and Cattel, R., (1998)] creates a new `Connection` object on-demand. However, a more efficient implementation is to use connection pooling which creates a linked-list of `Connection` objects, and creates new connections only when the pool is exhausted. While this approach was experimented with, it was found that connections obtained through the JDBC-ODBC driver were not persistent. That is, if the remote SQL server shut down, all connections in the pool would be lost. Therefore, for reasons of reliability, connections were created on-demand, and the code was written to minimize the creation of new `Connection` objects.

4.4 THE ARCHIVAL PROGRAM

The archival program is implemented in Java and performs all the functions described in section 4.1. The full source for the program can be downloaded from the CVS repository at http://wavelets.mit.edu/cgi-bin/cvsweb.cgi/archive/?cvsroot=Flagpole

The data archival program essentially consists of the following four threads:

1. A *data-listener thread*, which listens for data from the data acquisition server,
2. A *database-insert thread*, which inserts acceleration data into the database at the end of each minute,
3. A *database-purge thread*, which at the end of each hour, deletes data older than 24 hours, and
4. A *daemon thread*, which starts up and shuts down the data listener thread depending of whether the data acquisition server is active or not.

The following paragraphs describe the various threads in detail.

4.4.1 THE DATA-LISTENER THREAD

The archival program is implemented as a DataSocket subscriber (Figure 12) using the DataSocket JavaBean Beta 0.5 (http://www.ni.com/datasocket/ds_java.htm). All DataSocket operations are performed on an instance of a `DataSocket` class. Unlike the LabWindows/CVI implementation that uses a function pointer to handle event callbacks, the Java implementation uses a class implementing the `DSOnDataUpdateListener` interface.

```
23:     private static DataSocket socket; // data socket instance
 :
85:         socket = new DataSocket();
86:         socket.setURL("dstp://"+hostname+"/data");
87:         socket.setAccessMode(DSAccessModes.cwdsReadAutoUpdate);
88:         socket.setAutoConnect(true);
89:         socket.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
90:                 public void DSOnDataUpdate(DSOnDataUpdateEvent event) {
91:                     writeData(event);
92:                 }
93:             });
```

As shown in the listing, the DataSocket instance must first be passed the URL to which it must listen or publish. Then, it must be passed an access mode defined in the `DSAccessModes` interface (Connection modes are listed in Table 6 in Chapter 3). The `setAutoConnect` method is used to automatically connect to the DataSocket server specified. If this parameter is set to `false`, the DataSocket instance may be connected to the server by invoking the `connect` method. Finally, the DataSocket instance is

bound to an event listener. The DataSocket API triggers two events that can be captured: The DSOnDataUpdateEvent (which is triggered when new data is available) and DSOnStatusChangedEvent (which is triggered when the connection status changes). For each of these events, an interface is available, an instance of which can be associated with the DataSocket object. For the DSOnDataUpdateEvent, the DSOnDataUpdateListener interface is used and for the DSOnStatusChangedEvent, the DSOnStatusChangedListener is used. In the implementation shown above, the task of processing the data is performed by the writeData method, which is triggered once every 160 milliseconds.

As explained in section 3.3.2, data is published as a two-dimensional array of doubles. This data is then serialized into a byte stream and sent to the DataSocket server. The writeData method first converts this byte stream into a two-dimensional array again and determines the number of channels of data sent.

This is done in the following code segment:

```
144:    private static  void writeData(DSOnDataUpdateEvent event) {
145:        DSData fData = socket.getData();
146:        double readData[][];
147:        try {
148:            readData = fData.GetValueAsDoubleArray2D();
151:            channels = readData.length-1;
```

Here, the getData method is used to get the serialized data and the GetValueAsDoubleArray2D method is used to deserialize it into a two dimensional array. After that, it is quite straightforward to determine the number of channels (line 151).

The next task performed by this thread is to uniformly interpolate timestamp values given the start and end of the data block and write the interpolated timestamp and sample values to a text file. This is done in the following code segment:

```
192:        double interval = (readData[channels][0] - readData[channels][1]) / 15;
193:        double time = 0.00;
194:        double val=0.00;
195:
196:        for (int i = 0; i < 16; i++) {
197:            time = (double)(readData[channels][0]+i+interval);
198:            out.print(time + ",");
199:            for (int j=0;j<8;j++){
200:                if(j<channels)
201:                    val = readData[j][i];
202:                else
203:                    val = 0.00;
204:                if(j==7)
205:                    out.println(val);
206:                else
207:                    out.print(val+",");
208:
209:            }
210:        }
```

Here, `out` is a `PrintWriter` instance that writes data to a temporary buffer file. Note that if less than eight channels of data are published, the rest are set to zero by default. This is required because, when the bulk insert statement is executed, the text file must have an entry corresponding to each column in the table. Hence, accelerations corresponding to inactive sensors must be set to zero.

The data-listener thread also inserts the temperature into the database, but only once every 16 seconds, or 100 data cycles.

4.4.2 DATABASE-INSERT THREAD

This thread is triggered by the data-listener thread once every minute. Before this thread is called, the data-listener thread renames the buffer file according the elapsed minute and recreates another buffer for the next minute.

The database-insert thread then uploads this text file to the database using the bulk-insert feature in Microsoft SQL Server 7.0, as shown below:

```
227:        try {
228:            statement = "BULK INSERT flagpole_accel FROM \'"+textFile+"\' WITH
                                (FIELDTERMINATOR = \',\')";
229:            nRowsInserted = stmt.executeUpdate(statement);
230:            System.err.println(nRowsInserted+" rows inserted..."+ts.tstamp());
231:        } catch (SQLException e5) {
232:            e5.printStackTrace();
233:        }
```

It was found that the bulk insert feature, which uses the bcp program to copy data into the database, was substantially faster the batch update feature in JDBC.

After bulk-inserting the data into the database, the thread then adds the text file created into the zip file for that hour. At the end of each hour, a new zip file is created, and the purge thread is called, which is described next.

4.4.3 DATABASE-PURGE THREAD

The database purge thread simply deletes all records older than 24 hours. Instead of creating a statement every time, the purge operation is done using a stored procedure, sp_flagpole_accel_purge, shown below:

```
5:CREATE PROCEDURE sp_flagpole_accel_purge @start DECIMAL  AS
6:DELETE FROM flagpole_accel
7:WHERE time <= @start
8:DELETE FROM flagpole_accel_stats
9:INSERT INTO flagpole_accel_stats SELECT MIN(time) FROM flagpole_accel
```

The stored procedure also stores the time stamp of the least recent record in the database in flagpole_accel_stats. This is to ensure faster processing of queries, as explained in section 4.1.

The next section explains the working of the daemon thread that ensures reliable operation of the archive program.

4.5 MAINTAINING RELIABLE OPERATIONS

Consider a scenario in which the publisher shuts down for a short time and then starts up. Ideally, the subscriber must be able to detect this event and stop listening for data. However, once the publisher starts publishing data again, the subscriber must be able to resume data collection. Unfortunately, while there is some support for detecting lost

connections in the DataSocket API (using the `DSOnStatusChangedListener` interface), reclaiming lost connections is not very reliable, especially if the subscriber is down for a long time.

The daemon thread was implemented to overcome this shortcoming in the DataSocket API. It periodically checks to see if the data publisher is active. If the publisher dies for some reason, the thread disconnects the subscriber from the DataSocket, and once the publisher starts publishing again, the thread connects the subscriber back to the DataSocket. The daemon thread checks connection integrity by binding to a remote object on the server every 30 seconds. If a `RemoteException` [Sun Microsystems, (1999)] is thrown, it is assumed that the publisher is down, and the subscriber is disconnected. When the thread successfully binds to the remote object, the connection is re-established. Since the Java program on the remote server that exports the remote interface runs independently of the LabWindows/CVI program that publishes the data, it is quite possible (though quite unlikely), that the Java program dies while the publisher is running. The daemon therefore works well only if the remote machine itself is down, and not the individual services. To take care of this case, the archive program also uses the DataSocket APIs exception messages. However, as mentioned earlier, this mechanism is not very reliable for reclaiming lost connections.

The remote interface exported by the data acquisition server is actually used by applets to retrieve information from the database (explained in the next chapter). Therefore, ensuring reliable operations does not involve modifying the server at all. An alternate approach was tried before the one presented here was adopted. The daemon used the DataSocket object to read data from the server. If the read operation threw an exception, or if the time stamps in two consecutive read operations were the same, the daemon would assume a lost connection and disconnect automatically. While this was a very attractive proposition, it was extremely unreliable. That is, the read operation would continue to throw exceptions even if the publisher started broadcasting again. This was assumed to be due to a bug in the beta implementation of the DataSocket JavaBean.

The following code segment illustrates the implementation of the daemon.

```
105:    Thread daemon = new Thread(new Runnable(){
106:            public void run()
107:            {
108:                boolean flag=false;
109:                RMIInterface daemonInterface;
110:                while(true){
111:                    try{
112:                        daemonInterface = (RMIInterface)
                                             Naming.lookup("//"+hostname+"/RMI");
113:                        if(!flag){
114:                            socket.connect();
116:                            flag=true;
117:                        }
118:                    }catch(Exception e1){
121:                        if(flag){
122:                            socket.disconnect();
123:                            flag = false;
124:                        }
125:                    }finally{
126:                        try{
127:                            Thread.currentThread().sleep(30000);
128:                        }catch(InterruptedException e){
129:                        }
130:                    }
131:                }
132:            }
133:        });
```

4.6 CUSTOMIZING THE ARCHIVAL PROGRAM

Like the CVI data acquisition program, the archival program can also be customized using XML-like tags. These are summarized in the following table.

TAG	PURPOSE	EXAMPLE
<DATA>	Data file directory	`C:\Data\`
<JARPATH>	Location of `jar.exe`	`C:\jdk1.3\bin\jar.exe`
<LOGFILE>	File for logging messages (default: `stderr`)	`logs.txt`
<RMISERVER>	Host exporting remote interface	`fieldpoint.mit.edu` (default)

Table 9: Configuration for the Archival Program

CHAPTER 5. DATA RETRIEVAL AND VISUALIZATION

5.1 SOFTWARE FOR DATA RETRIEVAL

The previous two chapters described the data collection and archival processes in detail. This chapter discusses the various means by which real-time and archived data can be accessed. While the focus is primarily on writing Java clients, the next section briefly discusses how LabWindows/CVI programs can be written to access data via DataSockets. Only a few of the applications written to access real-time and archived data are described. More examples can be found in [Greene, D.C., (2001) and Nelson, J.M., (2001)] and on the project website, http://flagpole.mit.edu/realtime.html.

5.2 ACQUIRING REAL-TIME DATA USING LABWINDOWS/CVI

This section briefly describes the steps involved in implementing a DataSocket subscriber in LabWindows/CVI. The following paragraphs describe the various steps to be taken to plot the accelerations along the Y and Z axes of an accelerometer, which are published by the data acquisition program as described in section 3.3.

5.2.1 STEP 1: CREATING THE USER INTERFACE FILE

In LabWindows/CVI, user interfaces are created using the user interface editor, and saved with a .UIR extension. Each GUI component can be assigned a handle and can be bound to an event callback function. The following figure shows all the components in the user interface for a simple DataSocket subscriber.

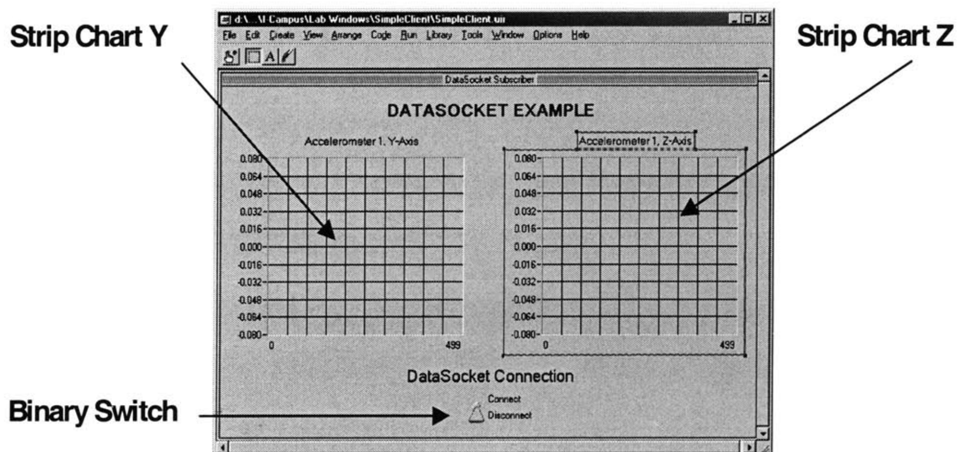


Figure 15: User interface for a DataSocket subscriber

5.2.2 STEP 2: IMPLEMENTING CALLBACKS

In this example, two types of event callbacks must be generated. The first is the event handler for the switch, and the second is the event handler for the DataSocket. The event handler for the switch is implemented as follows:

```
30:int CVICALLBACK SwitchCallback (int panel, int control, int event, void *callbackData,
                                   int eventData1, int eventData2)
31:{
32:   static int SWITCH_IS_ON = OFF;
33:   switch (event){
34:      case EVENT_COMMIT:
35:         SWITCH_IS_ON = SWITCH_IS_ON?OFF:ON;
36:         if(SWITCH_IS_ON){
37:            if(DS_Open ("dstp://fieldpoint.mit.edu/data",DSConst_ReadAutoUpdate,
                          DSCallback, NULL, &handle) < 0){
38:               printf("Error in getting connection! Bailing out...");
39:               exit(-1);
40:            }
41:         }
 :
```

When the switched it turned on, the DS_Open function is called, and a callback, `DSCallback` is passed to it. The connection mode is set as `ReadAutoUpdate` (Table 6), so the callback is triggered every time new data is published. If the connection is successfully opened, a non-zero reference to the DataSocket, `handle`, is returned.

The callback for the DataSocket needs to get the data as a two-dimensional array and plot all the elements in the first two rows (which correspond to the accelerations along the Y and Z axes for the first accelerometer). This can be done as follows:

```
53:void CVICALLBACK DSCallback (DSHandle dsHandle, int event,void* callbackData)
54:{
55:    int i,j,status;
56:    static int dim1,dim2;
57:    switch(event){
58:       case DS_EVENT_DATAUPDATED:
59:          if((status = DS_GetDataValue (handle, CAVT_DOUBLE | CAVT_ARRAY,
                                          vals, 96*sizeof(double), &dim1, &dim2)) < 0)
60:             return;
61:          else
62:             for(i=0;i<dim2;i++){
63:                PlotStripChartPoint (panelHandle, PANEL_STRIPCHART_Y, vals[0][i]);
64:                PlotStripChartPoint (panelHandle, PANEL_STRIPCHART_Z, vals[1][i]);
65:             }
66:             break;
67:    }
68:    return;
69:
70:}
```

The data is read into a two-dimensional array of doubles, `vals`, which has 6 rows and 16 columns(see section 3.3). The `DS_GetDataValue` function takes as its parameters, the DataSocket handle (`handle`), the type of data (array of doubles), the buffer into which the data is to be stored (`vals`), the size of the buffer (`6*16*sizeof(double)`) and the size of the array (`dim1` rows by `dim2` columns). The last two parameters, which are passed by reference, contain the actual dimensions of the incoming data. These might be different from the dimensions of the buffer used to store the data. The data is then plotted onto the stripcharts (`PANEL_STRIPCHART_X` and `PANEL_STRIPCHART_Y`) using the `PlotStripChartPoint` function. The following figure shows the program in operation.

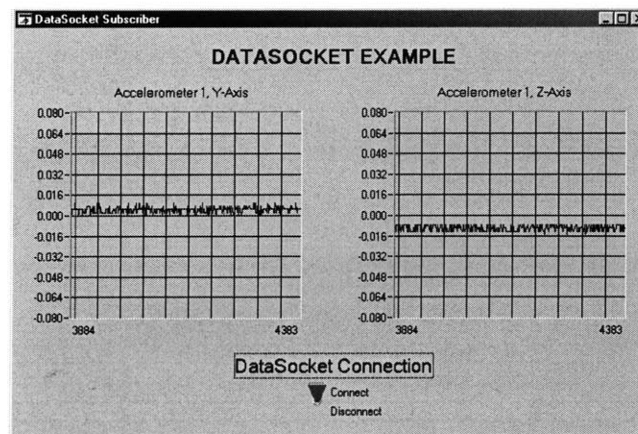


Figure 16: LabWindows/CVI DataSocket Client

The complete source code for this example can be downloaded from the CVS repository (http://wavelets.mit.edu/cgi-bin/cvsweb.cgi/simpleclient/?cvsroot=Flagpole).

While the LabWindows/CVI environment is very convenient for prototyping (the entire program in this case was less than 100 lines), deploying applications on other machines is very cumbersome because the CVI runtime must be installed on each client machine. Moreover, CVI applications cannot be conveniently accessed via a browser. Therefore, most of the applications written to access the data were written in Java. These are described next.

5.3 DISPLAYING REAL-TIME DATA IN JAVA

A Java client for a DataSocket server can be written in a manner similar to the data-listener thread in the archival program as discussed in section 4.4.1. Since Java does not have any built-in plotting classes, a custom plotting class, `GraphPanel` was implemented to visualize data. This is an abstract class, which is extended by two sub-classes, `GraphControl` and `StripChart`. While the former is used for plotting static data, the latter is used for plotting dynamically changing data in real-time. The `StripChart` class closely mimics the behavior or the strip chart control in LabWindows/CVI, but has additional functionality like auto-scaling and easy customization using the `GraphFormatter` helper class. Apart from being able to plot data in applets and applications, classes derived from `GraphPanel` can also output their plots as PNG images. This is useful for creating static snapshots of the plot, for instance, within servlets. While the `StripChart` class can plot real-time data in various ways, it prints only the sample numbers in the X-axis. When time stamps have to be printed in the X-axis, an instance of the `TimeStripChart` class can be used. This class takes a `SimpleDateFormat` as a constructor argument and prints the formatted time stamp in the X-axis for every sample. The following paragraphs illustrate the steps involved in displaying real-time acceleration data. These steps are also available online in the DataSocket tutorial, http://flagpole.mit.edu/applets/DataSocket/DS-HOWTO.html, which was written during the initial months of the project to enable other students to develop applets which process real-time data.

5.3.1 STEP 1:IMPORT PACKAGES AND INSTANTIATE DATASOCKET

The DataSocket API resides in the `natinst.msl.datasocket` package. The following code imports the library and instantiates a DataSocket instance.

```
6:import natinst.msl.datasocket.*;
7:
8:public class  DataSocketReader extends JApplet {
9:     private DataSocket dsHandle = new DataSocket();
```

5.3.2 STEP 2: ADD DATA UPDATE LISTENER

The next step is to add a data update listener, as explained in section 4.4.1. This is done as follows:

```
51:       dsHandle.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
52:               public void DSOnDataUpdate(DSOnDataUpdateEvent e) {
53:                   plotData(e);
54:               }
```

5.3.3 STEP 3: CONNECTING TO THE DATASOCKET

Connecting to the DataSocket involves providing the DataSocket instance with a URI and connection mode, as shown:

```
36:       dsHandle.connectTo("dstp://flagpole.mit.edu/data", DSAccessModes.cwdsReadAutoUpdate);
```

The connection mode is chosen as `ReadAutoUpdate` so that the event handler is triggered every time data becomes available.

5.3.4 STEP 4: IMPLEMENTING THE EVENT HANDLER

The event handler simply has to deserialize the data stream sent by the DataSocket. This is a two-step process. The first step is to get the serialized `DSData` object, and the second step is to deserialize it to a two dimensional array. If the type of data sent by the subscriber in not known *a priori* different data types can be derived using a `try-catch` block as explained in the tutorial. Once the array is obtained, it can be plotted using the `update` method of the `TimeStripChart` class. This is illustrated in the following segment:

```
87:     private void plotData(DSOnDataUpdateEvent e)
88:     {
89:         DSData fData;
90:         double [][] readData;
91:         fData = dsHandle.getData();
92:         try {
93:             readData = fData.GetValueAsDoubleArray2D();
94:             for(int i=0;i<16;i++){
95:                   stripChart.update(0,readData[0][i]);
96:             }
97:         }catch(DSDataException e1){
100:         }
101:     }
```

Source code for a full DataSocket reader implementation can be found in the CVS repository:

http://wavelets/cgi-bin/cvsweb.cgi/applets/datasocketreader/?cvsroot=Flagpole

5.4 ACQUIRING ARCHIVED DATA USING RMI AND JDBC

Section 4.3 described how the JDBC API along with a third-party driver can give Java programs access to various tabular data structures. Therefore, it would seem to be quite trivial to write an applet or an application using JDBC to access the database. However, this is not so for a number of reasons.

First, giving clients unrestricted access to the database can be very detrimental to its performance. For instance, if the client requests an unreasonably large number of records, the database could be tied in processing just one request and other clients might get locked out. It might also be possible for a malicious client to affect the integrity of the database by sending it malformed queries.

Second, deploying applets that access archived data is very difficult for a number of reasons. For instance, since the data acquisition server and the database server are different machines, and since an applet can make network connections only to the computer on which it is hosted, it can 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. Obviously, this is a very cumbersome procedure.

Finally, acceleration data older than a day must be extracted and placed into the database before queries can be made on it. This task must be implemented by every client, which would result in a lot of code duplication.

To make accessing archived data easier without affecting the integrity of the database, a set of remote interfaces were implemented using the Java Remote Method Invocation API [Sun Microsystems, (1999)]. RMI is a framework for calling methods against distributed Java objects. In principle, it is not very different from protocols such as RPC (Remote Procedural Call) or CORBA (Common Object Request Broker Architecture). However, unlike RPC, Java RMI is object-oriented, so it is possible to get reference to remote objects themselves instead of just executing remote methods. Moreover, unlike CORBA, Java RMI supports distributed garbage collection. The disadvantage of using RMI is that it is possible to invoke methods only on remote Java objects, since objects are serialized (converted into a stream of data) in a Java-specific binary format. Protocols such as SOAP (Simple Object Access Protocol), which use XML for serializing data, can be used to invoke methods on remote objects written in any language. Figure 17 shows how an applet hosted on the data acquisition server gets access to the database using RMI.

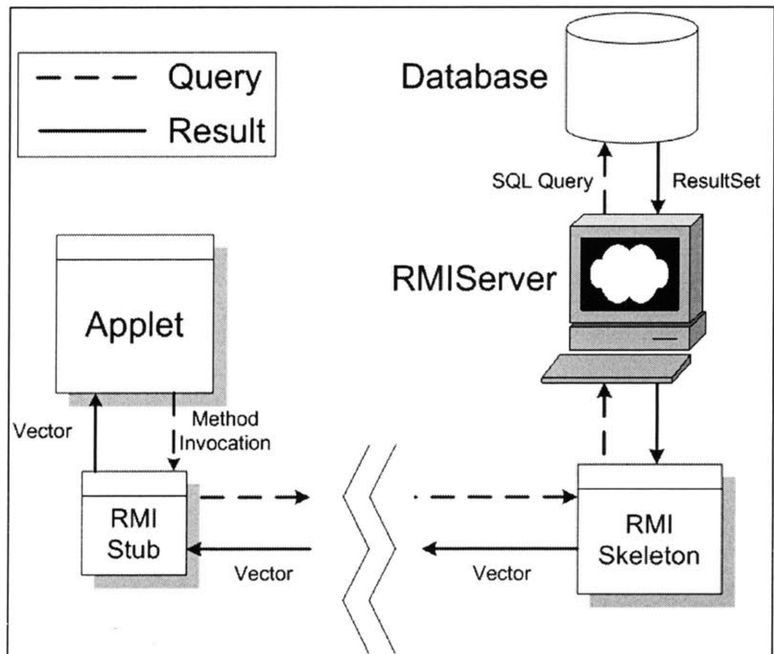


Figure 17: Database Access using RMI

As illustrated in the figure, a method invocation on a remote side occurs not on the actual remote object (the RMI Server, which queries the database), but on a proxy, know as the stub. The stub takes care of serializing the request and passing it on to the remote object. In the remote object, the method invocation is deserialized and a SQL query is generated after validating the request. The query is then executed on the remote database using JDBC. The `ResultSet` instance returned by the query is packed into a `Vector` and sent back to the stub. *Note that the `ResultSet` instance returned by JDBC is not serializable and hence cannot be directly sent across the network.*

The stub gets a reference to a remote object through a registry running on the remote machine. More details on how data is passed back and forth between the stub and the skeleton can be found in the RMI specification [Sun Microsystems, (1999)].

As shown in Figure 17, an RMI application consists of four components:

1. A Server which implements a remote interface,
2. A Skeleton which deserializes remote method requests, executes them on the server and sends the data back by serialization,
3. A Stub which serializes client method invocations, sends them to the skeleton, and returns the result to the client, and
4. A Client which invokes a method on a remote object

In addition, a registry is also needed on the server side that hands out references to objects exported by the server. However, as shown later, the registry may be embedded within the server itself.

The next few paragraphs illustrate the steps involved in creating all the components of a RMI client-server pair.

5.4.1 STEP 1: DECLARING THE INTERFACE

The first step is to declare the interface that the remote object must implement. While an object can implement other methods, only those declared by the remote interface can be invoked remotely by a client. All remote interfaces must inherit from the `Remote` interface in RMI, and all methods declared in remote interfaces must throw a

RemoteException. The following segment shows a part of the declaration of the interface used for communicating with the database.

```
15:public interface RMIInterface extends Remote {
16:    Vector getTemp(java.util.Date start,java.util.Date end)
            throws RemoteException, SQLException, NullPointerException;
17:    Vector getAccel(int channel,java.util.Date start,java.util.Date end)
            throws RemoteException, SQLException, IOException, NullPointerException;
```

Notice that the interface provides means only for retrieving acceleration and temperature data and not modifying it. Moreover, these remote methods return only the data values and not the associated time stamps. This is because queries such as temperatures during the last 24 hours or accelerations during the last minute may not need time stamps for all data points.

5.4.2 STEP 2: IMPLEMENTING THE INTERFACE

The next step is to implement the server. Apart from providing an implementation, the server accessing the database must also startup the RMI registry, which is actually responsible for handing out references to remote objects. If this were not done, the registry program, rmiregistry would have to be run independently as a service.

To simplify the creation, export and invocation of remote objects, the RMI API provides two abstract classes, RemoteObject and RemoteServer (which inherits from RemoteObject). Two concrete base classes, UnicastRemoteObject and Activable are then derived from these classes to provide a mechanism for transient and persistent objects (i.e., objects whose state can be shared by different JVM instances) respectively.

Since the RMI server need not be persistent (i.e. remote references need to be valid only for one JVM instance), it is derived from UnicastRemoteObject. The following code segment shows the relevant portions of the constructor for the RMI server.

```
18:public class RMIServer extends UnicastRemoteObject implements RMIInterface {
19:     private static Registry reg;
 :
35:     public RMIServer() throws RemoteException, ClassNotFoundException, SQLException {
36:         super();
37:         try{
38:             reg = LocateRegistry.createRegistry(1099);
39:         }catch(RemoteException e1){
```

Notice that the constructer to the base class must be invoked (Line 36) otherwise the remote object would not be automatically exported. An alternative approach would be to not extend the `UnicastRemotObject`, but to use the `exportObject` method in the `UnicastRemoteObject` class and manually export the remote object. Lines 37 to 39 are used to create a new RMI registry, or if one exists in the standard RMI port, 1099, use that instead.

Next, the remote interfaces defined earlier must be implemented. Getting temperatures is relatively straightforward. The method simply has to execute a `select` query on the database, get the result set, pack it into a vector, and return it to the client. These steps are illustrated in the following segment:

```
111:     public Vector getTemp(java.util.Date start,java.util.Date end)
                 throws RemoteException, SQLException, NullPointerException
112:     {
 :
119:         temp = conTemp.createStatement();
121:         ResultSet rs = temp.executeQuery("select time,temp1 from flagpole_temp"+
                                          "where time >= "+start.getTime()+
                                          " and time <= "+end.getTime());
 :
125:         while(rs.next()){
126:             eventTimes.add(new Long(rs.getLong(1)));
127:             eventData.add(new Double(rs.getDouble(2)));
128:         }
 :
134:         return eventData;
135:     }
```

Getting accelerations is quite similar, except now the method has to take care of uploading data from text files if necessary. This is handled in the following segment:

```
137:     public Vector getAccel(int channel,java.util.Date start,java.util.Date end)
            throws RemoteException, SQLException, IOException, NullPointerException
138:     {
 :
147:         if (start.before(minDate)) {
148:             if (end.before(minDate)) {
150:                 jarReaderInterface.extract(start, end);
151:             } else {
153:                 jarReaderInterface.extract(start, minDate);
154:             }
155:         }
 :
```

The minimum date, `minDate` is obtained from the `flagpole_accel_stats` table before the query is made, as shown in Figure 14. The rest of the steps (i.e. generating and executing the query, packing data into vectors, etc) are similar to the ones illustrated for the previous segment and hence are not described.

5.4.3 STEP 3: GENERATING STUB AND SKELETON CLASSES

The stub and skeleton classes are generated using the `rmic` compiler. The stub file and the interface are packed into a jar file to be used by the client and stored in a common library directory. The exact steps can be found in the build file for the project in the CVS repository (http://wavelets/cgi-bin/cvsweb.cgi/cviserver/build.xml?cvsroot=Flagpole).

5.4.4 STEP 4: BINDING TO THE REGISTRY

The final step on the server side is to bind to the registry (thereby enabling the object to be visible to remote clients). This is done by the `bind` method in the `Naming` class, which takes as its arguments, the URL under which the remote object is exported, and the remote object itself. For the RMI server, this is done in the following code segment:

```
365:         String name = "//localhost/RMI";
366:
367:         try {
368:             RMIInterface server = new RMIServer();
369:             Naming.bind(name, server);
 :
```

5.4.5 STEP 5: CUSTOMIZING AND EXECUTING THE RMI SERVER

The RMI server can be customized using tags. These are shown in Table 10.

TAG	PURPOSE	EXAMPLE
<RMIHOST>	Server hosting the JarReader service	flagpole.mit.edu (default)
<LOGFILE>	Log file name (default stderr)	Logs.txt

Table 10: Configuration for the RMI server

As mentioned earlier (Table 5), the RMI server runs on the data acquisition machine. Therefore, it is convenient to spawn it off and shut it down from the LabWindows/CVI data acquisition program. This is done using the LaunchExecutableEx and TerminateExecutable commands in LabWindows/CVI, as shown below:

```
                              START UP
83:     sprintf(command,
               "java -Djava.security.policy=permit edu.mit.flagpole.cviserver.RMIServer");
84:     result = LaunchExecutableEx (command, LE_SHOWMINIMIZED, &rmiServer);

                              SHUTDOWN
105:    result = TerminateExecutable (rmiServer);
```

The LaunchExecutableEx command returns an integer handle, rmiServer if the program is successfully spawned.

5.4.6 STEP 6: IMPLEMENTING A CLIENT

One of the advantages of a distributed method invocation is that the client does not have any inkling about the implementation of the remote method; all it needs to know are the method signatures, which are available through the interface. In the context of getting data from a remote database, the client does not need access to the database by itself. Therefore, tasks like configuring the data source, getting connections and validating inputs are all avoided.

Invoking remote methods involves two steps: Getting a remote reference, and invoking a method against it. A remote reference can be obtained using the lookup method Naming class on the client side. This method looks up the registry on the server and returns a reference to a generic Remote object exported using the supplied URL. This

reference can then be downcast as an RMIInterface object and the appropriate methods can be invoked on it. This is shown in the following block:

```
 62:           myInterface = (RMIInterface)Naming.lookup(rmiServer);
  :
323:           if(temperature)
324:               dataSet = myInterface.getTemp(start,stop);
325:           else
326:               dataSet = myInterface.getAccel(chooser.getSelectedIndex(),start,stop);
  :
```

Here, rmiServer is a string containing the URL of the remote object ("//fieldpoint/RMI") and start and stop are the beginning and end of the query period.

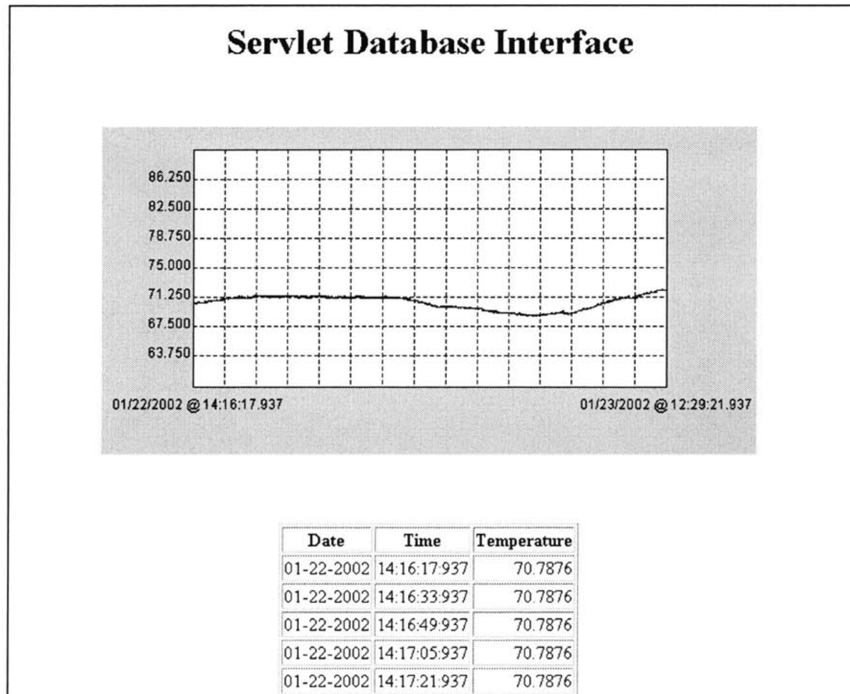
A full implementation of this client can be found on the real-time applets page, http://flagpole.mit.edu/databasedemo_frames.html.

5.5 INTERFACING A SERVLET TO THE DATABASE

One of the disadvantages with using an applet interface to the database is that most browsers do not support the version of JVM necessary to run applets written with Java 2. Hence, each client must download a full implementation of the Java Runtime Environment. Even with this in place, loading applets over networks is very slow. Moreover, due to file access restrictions, saving data generated by untrusted applets is not permitted. Hence, a servlet interface to the database was also implemented. This interface allows one to present the data as plain text, which can be saved and then loaded into programs such as MATLAB. The servlet can also output the results as HTML, along with a static plot of the data for convenient inspection. The front end to the servlet is coded in JSP, and can be accessed at http://flagpole.mit.edu/jsp/DBInterface.jsp. This JSP file sends the query to the DBInterfaceServlet that retrieves the records from the database and generates either a plain text or a HTML table.

When a static image of the data is to be generated, the data retrieved from the database is passed to another servlet (GeneratePNG) using a static method. The output from this servlet is then embedded as an image by the DBInterfaceServlet. This additional step is necessary because each servlet can output data of only one MIME type. In this

particular case, the servlet that outputs HTML (MIME type: text/html) cannot output images (MIME type: image/png). The `GeneratePNG` servlet uses the `encodeImageAsPNG` method in the `TimeStripChartClass` to generate a PNG image of the data on the fly. The following figure shows the temperature variation over a 24-hour period (5400 data points), which is retrieved from the database by the `DBInterfaceServlet`.



Date	Time	Temperature
01-22-2002	14:16:17:937	70.7876
01-22-2002	14:16:33:937	70.7876
01-22-2002	14:16:49:937	70.7876
01-22-2002	14:17:05:937	70.7876
01-22-2002	14:17:21:937	70.7876

Figure 18: Accessing the database through a servlet

CHAPTER 6. EDUCATIONAL TOOLS FOR THE VIRTUAL LABORATORY

6.1 EDUCATIONAL GOALS

As mentioned in Chapter 1, one of the goals of creating the virtual laboratory was to enhance the comprehension of concepts in structural dynamics, sensor technology and signal processing using a real-world structural system as a laboratory model. This chapter describes some of the software tools that were developed for this virtual laboratory with the above-mentioned goals in mind.

These educational tools fall into two categories. The first category consists of applets that illustrate a particular concept using numerical simulations. The second category consists of applets that process real data in different ways to illustrate concepts in structural dynamics and signal processing. In addition, it is also possible for a student to write his or her own programs to access and process real-time or archived data by modifying the example programs written in Java and Lab Windows/CVI.

In this chapter, these educational tools are described in more detail.

6.2 APPLETS THAT UTILIZE REAL-TIME DATA

Applets that process real-time data can be found in the real-time software page, http://flagpole.mit.edu/realtime.html. In this section, one of the applets which performs wavelet decomposition of the acceleration in real-time is described in detail.

6.2.1 REAL-TIME WAVELET DECOMPOSITION APPLET

This applet uses acceleration data to identify the modes of vibration of the flagpole using two methods. The first method determines the dominant frequency by computing a fast Fourier transform of 1024 data samples. The second method computes a fast wavelet transform of the acceleration data and locates the frequency subband in which coefficients with maximum energy occur.

The main goal of this applet is to compare the two methods in terms of their accuracy and processing delay.

The first method (based the fast Fourier transform) has very good frequency resolution (it gives the dominant mode exactly) but needs 1024 samples before it can produce an output. In contrast, the method based on the wavelet transform is almost instantaneous in producing outputs, but does not have very good frequency resolution (i.e., it gives only the subband in which the coefficients with maximum energy occur). This trade-off is clearly visible in the following figure. While the FFT gives the dominant mode of vibration as 0.586 Hz, the wavelet decomposition only shows that the dominant mode lies in the 0 to 6.25 Hz subband. Note that the sampling rate is 100 Hz, and hence the frequency levels c0, d0, d1 and d2 in the figure correspond to 0 to 6.25 Hz, 6.25 to 12.5 Hz, 12.5 to 25 Hz and 25 to 50 Hz, respectively.

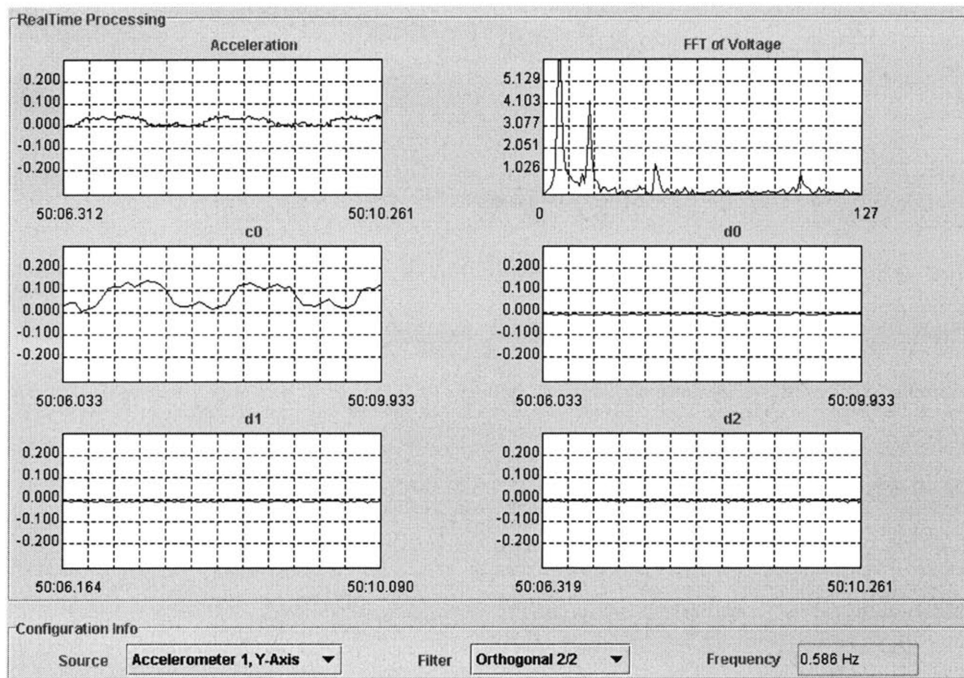


Figure 19 Identification of modes of vibration

The wavelet decomposition applet can be easily extended to demonstrate other applications in signal processing. One such extension is the removal of noise from the acceleration data. Here again there are two possible methods for removing high frequency noise from the data. One possibility is to compute the Fourier transform of a large number of coefficients and then remove those high frequency components whose magnitude is below a certain threshold. However, this again requires substantial delay before the output can be produced. An alternative approach is to transform the

acceleration data into the wavelet domain and zero out those coefficients in the high frequency subband that fall below a certain threshold. This approach can be used to produce a near real-time denoised signal from the acceleration data. A demonstration of such a denoising applet can be found at http://flagpole.mit.edu/denoise_frames.html.

Another extension to this applet is in archiving the acceleration records. Since most of the energy of the wavelet coefficients is concentrated in the low frequency subbands, the acceleration data can be efficiently archived by first representing the coefficients in the lower frequency bands using a larger number of bits and those in the higher frequency bands using a smaller number of bits. Upon quantization, many of the high frequency coefficients will be rounded to zero and hence can be efficiently compressed using runlength encoding. Details about such an implementation can be found in [Nelson, J.M., (2001)]. This particular implementation computes the wavelet coefficients using the lifting scheme and stores the coefficients in zip files using the `ZipOutputStream` class in the `java.io` package, which automatically handles the Huffman coding and runlength encoding steps.

6.3 SIMULATION APPLETS

Applets that do not use real-time data but demonstrate concepts using numerical simulations can be found at http://flagpole.mit.edu/education.html. In this section, two of the simulation applets are described. Description about other applets can be found in [Greene, D.C., (2001)] and [Schlingloff, J. (2001)].

6.3.1 BASELINE CORRECTION USING WAVELET FILTERS (APPLET BY DAVID GREENE)

This applet demonstrates the use of wavelet filters for correcting drifts in numerical integration due to measurement errors in the acceleration data. While the seismic event by itself is a zero-mean process, due to inaccuracies in measurement, the accelerations do not have a zero mean (they usually have a constant offset). Due to this, the velocities and displacements obtained by integrating the acceleration exhibit linear and parabolic offsets respectively (this is shown in Figure 20).

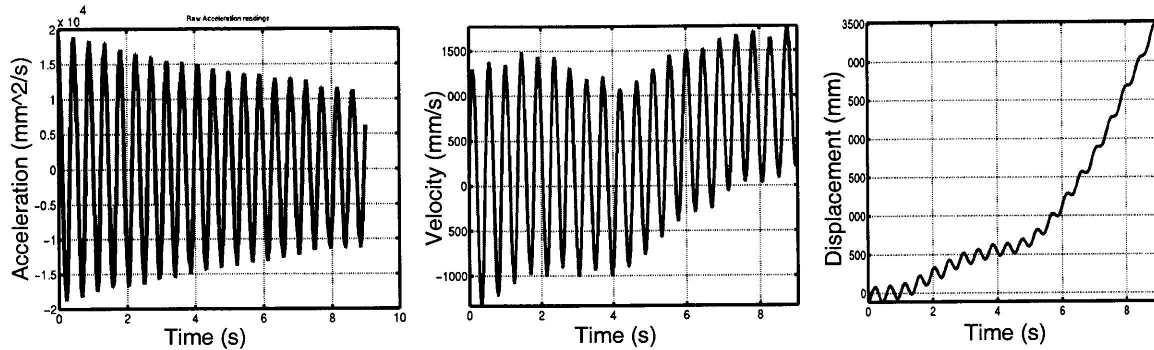


Figure 20 Offset errors in velocity and displacement

The technique commonly used for baseline correction is again based on the Fourier transform of the acceleration data. The constant offset in acceleration manifests itself as a non-zero value of the DC component in the Fourier transform domain. The signal is then baseline corrected using a band pass filter that removes the very low and the very high frequency components (which correspond to the offset and noise respectively).

However, the offset in the acceleration can also be corrected using wavelet decomposition of the acceleration data. This correction procedure is based on the polynomial approximation property of wavelet filters (Condition A). The parabolic envelope in the displacement data can be isolated in one of the low frequency subbands after a sufficiently large number of wavelet decomposition levels (provided the low pass filter has three or more zeros at π). This error can be removed from the displacement by ignoring the coefficients in this subband while reconstructing the signal.

This applet also superimposes the response obtained after drift correction with the theoretically predicted response and compares the two.

It must be mentioned however, that in reality the baseline correction procedure is not as straight forward since before and after the first numerical integration step the signal has to be denoised for the parabolic profile to manifest itself distinctly in the low-pass channel.

6.3.2 VIBRATION CONTROL USING A TUNED MASS DAMPER

A tuned mass damper is an energy dissipation device used in tall buildings to reduce deflections due to wind loading. Unlike a viscous damper that provides a damping force proportional to the velocity of vibration, a tuned mass damper is an inertial system. That

is, the damping force is provided by a mass which vibrates out of phase with the structure. The equations governing the response of the mass and damper can be found in Chapter 4 of [Connor, J.J., (2001)] (where the equations are derived based on relative motion) and under the "More Information" link in the applet (where the equations are derived in terms of absolute displacements).

One of the design issues for a tuned mass damper is providing an optimal damping ratio for the damper. The applet lets the user experiment with different parameters for the damper (mass, stiffness and damping) and "tune" it such that the building has a minimum response to periodic excitation. The applet also allows the user to simulate periodic ground excitation (which can occur for instance due to the presence of heavy machinery in the vicinity of a structure, or due to an earthquake).

The following figure shows a screen shot of the tuned mass damper simulation with only wind loading, excited at resonance.

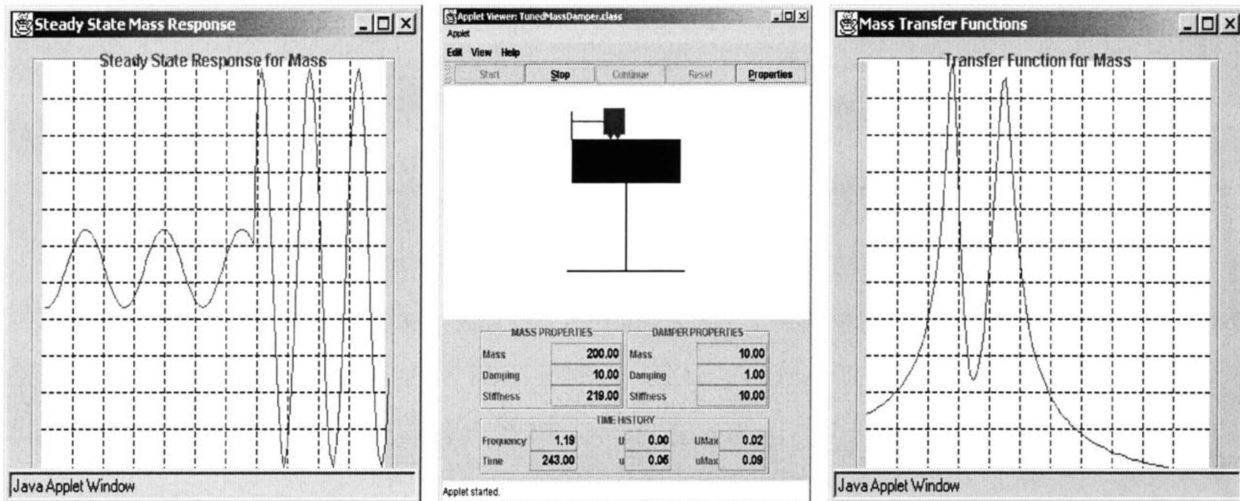


Figure 21 Screenshot of the tuned mass damper applet

6.4 USING THE SOFTWARE TOOLS IN AN ENGINEERING CURRICULUM

It has been observed that the use of remote laboratories has been most effective in courses that have substantial hands-on or experimental components. For example, the photovoltaic weather station is used for course 1.105 (Solid mechanics laboratory), the microelectronic web lab is used for course 6.012 (Microelectronic devices and circuits) and the chemical reactor web lab is used for course 10.302 (Transport processes), all of which have significant experimental content.

The software developed for the flagpole virtual laboratory is valuable in an educational setting because it allows different applications to be developed simply by extending the rich client application code base. As demonstrated in this chapter, applications can be written to explore different numerical and signal processing algorithms by testing them on real data. In addition to using the end products in the classroom, there is a significant educational benefit to using the software platform in a course with a significant programming component. Courses which have used the software tools in the past include 1.124 (Foundations of software engineering), 1.120 (Information technology M.Eng. project) and 1.130 (Information processing for engineering systems). The numerical simulations can also be used to supplement concepts covered in the classroom, and many of the educational applets have been used in courses such as 1.561 (Motion based design) and 1.030 (Civil engineering materials).

CHAPTER 7. CONCLUSIONS AND FURTHER WORK

7.1 SUMMARY AND CONCLUSIONS

As mentioned in section 1.2 (Objectives), the main goal of this virtual instrumentation project was to implement a prototype remote virtual laboratory for real-time structural infrastructure monitoring, which could eventually be extended to an entire metropolis to make the concept of the I-City practically realizable. Another goal was to implement a test platform for developing educational tools and programs that could be used to process data in real-time. Such a platform could be used for testing various numerical algorithms and expose students to the state of the art in information technology, sensor design and signal processing.

The software components in this project (both server and client side) were designed and implemented with the above-mentioned goals in mind. The architecture for server side data archival and retrieval has already been adapted with very few modifications to several other virtual laboratories. The monitoring system has been in operation continuously for the past two months and real-time and archived data are available on demand to clients anywhere on the Internet.

On the client side, a number of examples have been provided, which access and process real-time and archived data in a number of ways. These applications can easily be adapted in a classroom environment and many numerical and signal-processing concepts can be illustrated on real, instead of artificially generated, data.

While most of the original goals of the remote virtual laboratory have been met, there still remains a lot of work to be done, particularly in implementing wireless sensor networks, developing clients for different platforms and developing collaboration, assessment and educational evaluation tools. These are discussed in the next section.

7.2 FURTHER WORK

7.2.1 DEPLOYING WIRELESS SENSORS

Currently, data from the accelerometers is being acquired using a shielded underground cable. However, it is hoped that in the near future the connection between the sensors and the data acquisition server will be made wireless. During the course of the project, many wireless technologies were tested in conjunction with the FieldPoint monitoring system (as mentioned in section 2.4), but were found lacking in many ways. Crossbow Technologies has recently brought to market wireless sensor systems (CN1100) based on the Bluetooth standard that have a 100 m range and can sample at up to 1kHz. It is hoped that such wireless sensors will be deployed sometime in the near future. However, these systems are not compatible with the FieldPoint monitoring system, and hence, some of the server-side software would have to be re-written.

An alternative to the CN1100 is the second-generation wireless sensor network system known as MICA, also developed by Crossbow. MICA sensor boards can be used to create low power, large-scale sensor network systems, and have proved to be quite successful in several applications such as defense and environmental monitoring. Hence, these are ideal for infrastructure monitoring efforts, such as the flagpole virtual laboratory.

7.2.2 MORE EFFICIENT DATABASE ACCESS

The archival system returns acceleration data acquired during the last 24 hours very quickly since it is available in the database. However, data older than a day has to be extracted from zip files and uploaded into the database before queries can be made. While this strategy works in practice, it is extremely slow.

A better solution would be to store only the compressed wavelet coefficients of older data in the database. When a query for older data is made, the wavelet coefficients could be quickly extracted and approximate sample values could be reconstructed on the fly. Such a scheme would reduce the data to manageable amounts and hence permit faster retrieval of older data.

7.2.3 DEVELOPING CLIENTS IN MICROSOFT .NET

All the web accessible clients implemented for the current project are written in Java. However, it would also be beneficial to develop clients in the Microsoft .NET environment, which has emerged as a popular platform for developing web services. National Instruments is on the verge of releasing the Component Works libraries for the .NET SDK. This planned release includes a .NET version of the DataSocket API that would greatly simplify the task of implementing clients accessing real-time data.

7.2.4 EXTENSION TO OTHER REMOTE LABORATORIES

During the design and implementation of the virtual laboratory, it was ensured that the software would scale well to other monitoring projects. Hence, a natural extension of this project would be to monitor other physical infrastructure. One such effort currently under way is the Smart Wells project that monitors various hydrological parameters in wells around the MIT campus.

Another possible extension could be to incorporate sensors in different locations as part of the current laboratory itself. Such an extension could be easily accomplished since it would involve very few changes in the pre-existing code base.

7.2.5 EDUCATIONAL EVALUATION AND ASSESSMENT

Educational evaluation and assessment are important components that need to be incorporated into the education framework of our virtual laboratory. Data from assessment studies would give us a clear picture of the utility of the educational modules (simulations as well as real-time applets), which could then be used to improve the existing applets or develop new programs and tools to illustrate other concepts. Many web accessible laboratories (such as the microelectronics web laboratory and the heat exchanger laboratory) already have tightly integrated assessment components as part of their experiments and have used the results from these studies to improve their experiments. Such an assessment tool would certainly go a long way in enhancing the educational benefits of the flagpole virtual laboratory.

APPENDIX A. REFERENCES

[1] Bluetooth Special Interest Group, (2001), *"Specifications of the Bluetooth System"*, (http://www.bluetooth.com/pdf/Bluetooth_11_Specifications_Book.pdf)

[2] Boriello, G., and Want, R., (2000), *"Embedded Computation Meets the World Wide Web"*, Communications of the ACM, 43(5), pp 59-66.

[3] Campione, M., and Walrath, K., (1998), *"The Java Tutorial, 2nd Edition"*, Addison-Wesley Pub. Co., Reading, MA.

[4] Chopra, A.K., (1995), *"Dynamics of Structures: Theory and Application to Earthquake Engineering"*, Prentice-Hall, Upper Saddle River, NJ.

[5] Chui, C.C., (1998), *"Wavelets: A Mathematical Tool for Signal Processing"*, Society for Industrial and Applied Mathematics, Philadelphia, PA.

[6] Connor, J.J., (2001), *"Introduction to Structural Motion Control"*, (http://moment.mit.edu/r_textbook.asp).

[7] Crossbow Technology, (2000), *"Accelerometer Technology Training"*, Crossbow Technology, Inc., San Jose, CA,

(http://www.xbow.com/Support/Support_pdf_files/Accelerometer%20Technology%20Training.ppt)

[8] Doscher, J., (1995), *"Using iMEMS accelerometers in Instrumentation Applications"*, Application Note, Analog Devices, Norwood, MA, (http://www.analog.com/library/techArticles/mems/imems_accl.html).

[9] Gosling, J., et al., (2000), *"The Java Language Specification, 2nd Edition"*, Addison-Wesley Pub. Co., Reading MA.

[10] Greene, D.C., (2001), *"Sensor Technology and Application to a Real-Time Monitoring System"*, Master of Engineering. Thesis, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA, (http://flagpole.mit.edu/Theses/thesis_dcgreene.doc).

[11] Hamilton, G., and Cattel, R., (1998), *"JDBC™: A Java SQL API, Version 1.20"*, Sun Microsystems, Mountain View, CA, (http://java.sun.com/j2se/1.3/docs/guide/jdbc/spec/jdbc-spec.framce.html).

[12] Hung, E.S., and Zhao, F. (1999), *"Diagnostic Information Processing for Sensor Rich Distributed Systems"*, Proc. 2nd International Conference on Information Fusion, Sunnyvale, CA.

[13] Khazan, A.K., (1994), *"Transducers and Their Elements"*, Prentice-Hall PTR, Englewood Cliffs, NJ.

[14] Krishna, C.M., and Shin, K.G., (1997), *"Real-Time Systems"*, The McGraw-Hill Companies, Inc.

[15] Kumar, S., et al., (2002), *"Collaborative Signal and Information Processing in Micro-Sensor Networks"*, IEEE Transactions on Signal Processing, to appear.

[16] Moffat, R.J., (1997), *"Notes on Using Thermocouples"*, Electronic Cooling, 29, http://www.electronics-cooling.com/Resources/EC_Articles/JAN97/jan97_01.htm

[17] National Instruments, (1998), *"FieldPoint Benchmarks"*, National Instruments Corporation, Austin, TX (ftp://ftp.ni.com/support/fieldpoint/Server/fp16_rates.pdf).

[18] National Instruments, (1999a), *"Integrating the Internet into Your Measurement System: DataSocket Technical Overview"*, National Instruments White Paper No. 1680, National Instruments Corporation, Austin, TX, (http://www.ni.com/pdf/wp/wp1680.pdf).

[19] National Instruments, (1999b), *"FP-1600 User Manual"*, National Instruments Corporation, Austin, TX, (http://www.ni.com/pdf/manuals/322394a.pdf)

[20] National Instruments, (1999c), *"Sampling Rates of FieldPoint Modules"*, National Instruments Knowledge Base Document 1O3CJ7US, National Instruments Corporation, Austin, TX, (http://digital.ni.com/public.nsf/3efedde4322fef19862567740067f3cc/0d418c9097f14e99862567c300666568?OpenDocument)

[21] Nelson, J.M., (2001), *"Real-Time Data Collection and Archiving of Physical Systems"*, Master of Engineering. Thesis, Department of Civil and Environmental

Engineering, Massachusetts Institute of Technology, Cambridge, MA, (http://flagpole.mit.edu/Theses/thesis_nelsonj.doc).

[22] Pavlou, Y. (1999), *"Data Acquisition Fundamentals"*, National Instruments Application Note, Austin, TX, (http://www.chipcenter.com/benchtop/tn003.html).

[23] Pottie, G.J., and Kaiser, W.J., (2000), *"Wireless Integrated Sensor Networks"*, Communications of the ACM, 43(**5**), pp 51-58.

[24] Roush, W. (2001), *"Networking the Infrastructure"*, Technology Review, December, pp 38-42.

[25] Schlingloff, J. (2001), *"Development of Interactive and Real-Time Educational Software for Mechanical Principles"*, Master of Engineering. Thesis, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA, (http://flagpole.mit.edu/Theses/thesis_schlingl.doc).

[26] Sun Microsystems, (1999), *"Java ™ Remote Method Invocation Specification"*, Sun Microsystems Inc., Mountain View, CA.

[27] White, S., et al., (1999), *"JDBC™ API Tutorial and Reference, 2nd Edition"*, Addison-Wesley Publishing Co., Reading MA, (http://java.sun.com/docs/books/tutorial/jdbc/).