

Real-Time Data Collection and Archiving of Physical Systems

by

James M. Nelson

B.S. Mathematics, Computer Science
University of Sioux Falls, 2000

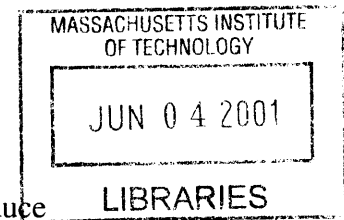
SUBMITTED TO THE DEPARTMENT OF CIVIL AND ENVIRONMENTAL
ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF ENGINEERING IN CIVIL AND ENVIRONMENTAL ENGINEERING
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2001

©2001 James M. Nelson. All rights reserved.

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



Signature of Author: _____
Department of Civil and Environmental Engineering
May 11, 2001

Certified by: _____
Kevin Amaratunga
Assistant Professor, Civil and Environmental Engineering
Thesis Supervisor

Accepted by: _____
Oral Buyukozturk
Chairman, Departmental Committee on Graduate Studies

Real-Time Data Collection and Archiving of Physical Systems

by

James M. Nelson

Submitted to the Department of Civil and Environmental Engineering
on May 11, 2001 in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Civil and Environmental Engineering

ABSTRACT

The I-Campus Flagpole Instrumentation Project aims to monitor, simulate, and visualize the Building 1 flagpole on the MIT campus. All of the initial work completed was on a prototype of the system. Sensors were placed on the prototype to measure its properties, namely acceleration, strain, and temperature. A data acquisition system, FieldPoint from National Instruments, has been adopted to collect information from the sensors and transmit it to data collection software. Experiments were made with wireless technologies such as WaveLAN and Bluetooth, but their use has been reserved for the next iteration of the project.

Once the data exists in software, it is made available for visualization tools in real-time, as well as archived for later analysis. Real-time data is provided through a broadcasting technology called data sockets by National Instruments. The archiving process includes short-term storage in a database and a handful of compression techniques for persistent storage. A client interface has been developed for client applications to interact with both real-time and archived data through a set of APIs. Client applications requiring data request from a common place, but the actual execution and processing is distributed between servers using RMI, based on the nature of the request.

Thesis Supervisor: Kevin Amaratunga

Title: Professor of Civil and Environmental Engineering

Acknowledgments

First of all, I would like to thank God for giving me the strength and perseverance needed to complete this thesis and program. I also want to thank my loving wife Angela for her patience and understanding throughout this year.

In addition, I want to thank my thesis advisor and project supervisor, Kevin Amaratunga, for his high standards and continued support. Special thanks to Raghunathan Sudarshan, for working with him made up much of the content in this thesis. Also, I want to thank the M.Eng IT group for all of their help and comic relief.

Lastly, I would like to thank my parents John and Joan, and my brothers Mike, Tom, and Dan for always challenging me and driving me to reach my fullest potential.

Table of Contents

1	Introduction	9
2	Flagpole Instrumentation Project.....	11
2.1	Flagpole Instrumentation Goals	12
2.2	System Overview	13
2.3	Hardware	14
2.4	Software	14
2.5	Visualization.....	15
2.6	Future Scope.....	16
3	Data Collection	18
3.1	Hardware	18
3.2	Data Communication.....	20
3.3	Software	23
4	Data Archiving.....	27
4.1	Database Management Systems	27
4.2	Archiving Process	31
4.3	Compression.....	33
5	Client Interface	42
5.1	Real-time Data.....	42
5.2	Archived Data	43
6	Conclusions	51
7	References	52
	Appendix A: Database Wrapper Class	53
	Appendix B: Archive Program	54
	Appendix C: Wavelet Decomposition Applet	58
	Appendix D: Real-time and Archive Data Visualization Applet	63
	Appendix E: Database to Data Socket Servlet.....	66
	Appendix F: RMI Server (i-city)	68
	Appendix G: RMI Interface (i-city).....	73
	Appendix H: JarReader Program	74
	Appendix I: JarReader RMI Interface	76

Table of Figures

Figure 2.1 System Overview	13
Figure 3.1 FieldPoint Module	19
Figure 3.2 Real-time Data Display.....	24
Figure 4.1 Data Model	28
Figure 4.2 Data Storage.....	31
Figure 4.3 Analysis Filter Bank	35
Figure 4.4 Haar Filter Bank Decomposition	36
Figure 4.5 Haar Filter Bank Decomposition with Lifting.....	37
Figure 4.6 Synthesis Filter Bank.....	39
Figure 4.7 Haar Filter Bank Reconstruction with Lifting	40
Figure 4.8 Wavelet Decomposition Applet.....	41
Figure 5.1 Client Interface Overview.....	42
Figure 5.2 Archive Retrieval	44
Figure 5.3 Archived Data “Behaving” Like Real-time	45
Figure 5.4 Archived Data Returned in One Bundle	46
Figure 5.5 Distributed System.....	48

1 Introduction

As we progress through the 21st century, the world is moving towards dynamic, intelligent structures. Information technology is appearing in locations and situations not thought of even ten years ago. The I-Campus initiative, a MIT-Microsoft collaborative project, is aimed at using information technology to improve education. One project in particular under the I-Campus umbrella, the Flagpole Instrumentation Project, is primarily concerned with making use of breakthrough technologies to monitor and simulate the properties of the Building 1 flagpole on the MIT campus in real time.

The purpose of this thesis is two-fold – to serve as documentation for the Flagpole Instrumentation Project and to explain the details of how to implement a real-time data collection and archiving solution for a physical system. I hope to educate the reader about the entire process of the design and implementation of the Flagpole Instrumentation Project – the failures as well as the successes. I will discuss the options that were available, the choices made, and the logic used to make some of the technical decisions in the project.

In the second chapter, an overview of the I-Campus Flagpole Instrumentation Project is given, paying special attention to the goals and ideals of the project. The chapter begins with an introduction to I-Campus and the Flagpole Instrumentation Project. The project is then divided into its three main components – hardware, software, and visualization. After that, each of these components is explained in detail. Finally, the future scope and challenges of the project are discussed.

Chapter 3 gives insight into the data collection process used in the Flagpole Instrumentation Project. The data collection process is divided into three parts – hardware, data communication, and software. The hardware is further subdivided into sensors and the data acquisition system, or the FieldPoint module for our project. The data communication section deals with the many issues surrounding the transfer of data

from the data acquisition system to the servers, including wireless attempts and problems, as well as the current setup. The software section is separated into the data acquisition server and data sockets.

The fourth chapter is primarily concerned with the data archiving tools used and techniques implemented in the Flagpole Instrumentation Project. First, a look is given into the different database management systems investigated in the project, as well as the database drivers available to interface with them. The data archiving process currently implemented is then discussed in detail, along with the trade-offs made and some rationale for the choices made. From there, the compression technique presently implemented in the project is explained, namely using the jar utility. Finally, an alternative compression method making use of wavelets and filter banks is discussed in detail.

Chapter 5 describes the interface developed for client applications to interact with the two major types of data – real-time and archived. A look is then given to the real-time data interface as well as a code example. Archived data retrieval is further subdivided into two categories; “mock” real-time and bulk upload. The problems and concerns that arise while trying to extract the data are explained, such as distributed computing and compressed data retrieval.

The conclusions and suggestions for the continuation of the project are discussed in chapter 6. Finally, the code for many of the programs discussed throughout the thesis is included in the appendices.

2 Flagpole Instrumentation Project

The physical infrastructure of the MIT campus provides an excellent setting for learning about advances in sensing, control, communications and information technologies. These technologies are key enablers of the highly instrumented physical environments of the future, which range from homes with networked appliances to entire digital cities in which buildings, highways, bridges and tunnels publish information about themselves and exhibit controlled responses to external influences.

In this multidisciplinary project, funded by Microsoft, the focus is on the instrumentation of a small portion of the MIT campus – namely the Building 1 flagpole – and building a “virtual” laboratory (also known as an I-Lab) around it. The goal is to develop a web-based information system to monitor the dynamic response of the flagpole in real time and to analyze, predict, and even control, perhaps in a virtual sense, its behavior. The Flagpole I-Lab will be subsequently used in the teaching of undergraduate and graduate courses on structural engineering. Hence, the system architecture must be both flexible, so that it can be adapted with relative ease to other scenarios such as transportation and the environment, and scalable, ideally up to the extent of an entire city.

The Flagpole I-Lab is one of several I-Labs that are being developed as part of the MIT-Microsoft I-Campus initiative. In return for the opportunity to participate in this high profile effort, students agree that the results of this project (including software) are subject to the terms of the MIT-Microsoft I-Campus agreement. This agreement aims to make the results of the I-Campus projects freely distributable, but does not preclude Microsoft from exercising commercial rights to the results. The concept of I-Campus is to develop a virtual environment where one can model real world systems. The goal is to take any physical system, such as a building or bridge, place sensors on it to monitor its behavior, and use the information to predict failure mechanisms and actively control the system.

I-Campus could create the opportunity to define a new role for the next generation of civil engineers. Also, vast new research and educational opportunities could put civil engineering in the forefront of modern technology. This opportunity would strengthen inter-departmental ties and be a unifying theme for the CEE department. Some of the goals of I-Campus include:

- Monitor real world systems
- Create an interface between engineering and information technology
- Develop an educational environment to promote higher learning for students
- Investigate and implement advances in monitoring technology
- Learn how to make effective use of virtual environments

2.1 Flagpole Instrumentation Project Goals

The goal of the Flagpole Instrumentation Project is to build a real-time monitoring system for the Building 1 flagpole on the MIT campus. This includes placing sensors on the outer surface of the flagpole to determine its mechanical and dynamic behavior. The data is streamed and archived from where it becomes available for monitoring software. In summary the basic steps include:

- Obtain and calibrate sensor devices for thermal, strain and position monitoring
- Devise a data collection and archiving solution
- Develop a visualization and monitoring platform that can be utilized in any real world system

The integration of physical and virtual learning spaces will add a new dimension to students learning in a broad range of areas, including:

- **Hardware:** sensors, data collection hardware, and wireless communications
- **Software:** data acquisition, data broadcasting technologies, database management systems, compression techniques, and simulation systems
- **Visualization:** nondestructive evaluation, educational applet creation, and visualization applet creation

The learning environment will serve as a shared remotely accessible laboratory, which will educate undergraduate engineers in systems level thinking and expose them to the exciting new possibilities afforded by information technology. In addition, specialized elements of the environment will feed into graduate education and research. The remote laboratory concept will also provide a portal to MIT for distance education partners, industry, alumni, and the outside world. The instrumentation of the Building 1 flagpole will be the initial focus in turning the physical MIT campus into a hands-on virtual laboratory.

2.2 System Overview

The system is composed of three major parts: hardware, software, and visualization. Figure 2.1 illustrates the general system overview.

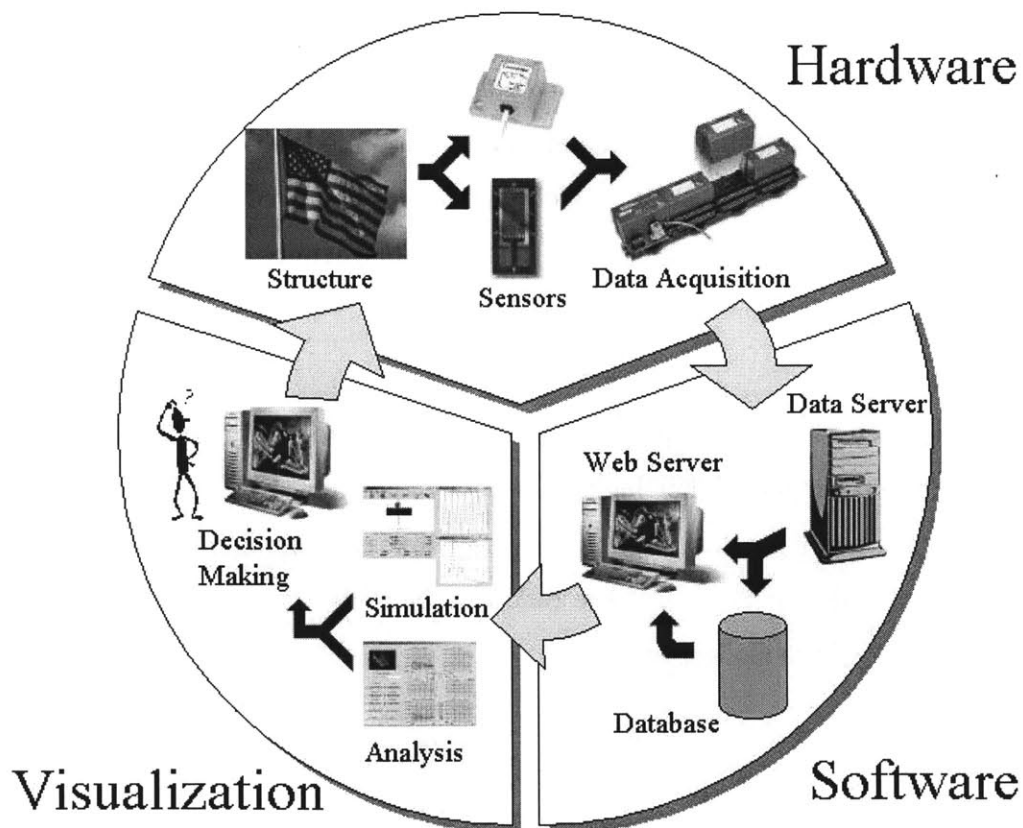


Figure 2.1 System Overview

Figure 2.1 displays the integrated trinity of the Flagpole Instrumentation Project – the combination of hardware, software, and visualization. The general flow of data is summarized as follows:

- A data acquisition system collects data from sensors on a structure
- A data server collects the data from the data acquisition hardware
- Data is archived in the database as well as in compressed format
- Information is made available to web clients via data sockets
- Visualization, analysis, and simulation applets illustrate system properties
- Decision-making tools allow dynamic control of structure

The Flagpole Instrumentation Project combines elements of software, hardware, and structural engineering, computer simulation, and design. The current setup is a small-scale, simple prototype for the testing of monitoring devices (strain gages, accelerometers, and thermocouples) and data acquisition modules. The model is an upright aluminum bar attached to a metal base. Such a model can be characterized as a single degree of freedom system, which simplifies the generated data because its properties are easily calculated using basic mechanical principles.

2.3 Hardware

The hardware used in the I-Campus Flagpole Instrumentation Project consists of sensors to measure the properties of the flagpole, a data acquisition system to collect the data from the sensors, and a handful of attempts with wireless communications to transfer the data to the data acquisition server. The hardware currently installed will be elaborated later in Chapter 3 – Data Collection.

2.4 Software

Software is probably the most loosely defined section of the Flagpole Instrumentation Project because it plays an important role in all of the stages of the project. There are three main areas of research undertaken by the software team:

1. **Data Collection** – establishing and optimizing the connection between the data acquisition hardware and the data server. The data collection techniques used in the Flagpole Instrumentation Project are discussed in detail in Chapter 3.
2. **Data Archiving** – investigating and implementing data archiving techniques such as database management systems and real-time compression. Chapter 4 provides an in-depth look at the many attempts at creating a robust data archiving system.
3. **Client Interface** – providing a simple interface for client applications to interact with real-time as well as archived data. The current interface between clients and data, whether real-time or archived, is explained in Chapter 5.

2.5 Visualization

The vision of the Flagpole Instrumentation Project is to build an integrated system to serve as the front end of a virtual lab, which will be a unique multi-disciplinary learning environment. With the web front end of the flagpole virtual lab, students could conduct traditional lab experiments through a virtual manner, namely modify physical parameters remotely and see the result immediately without the need of being in a “real” lab classroom.

One of the goals of the flagpole project is to develop Java applets to simulate the flagpole in real time as well as obtain information through archived data. A future goal is to develop an active control system, either virtually or physically. In an effort to fulfill these goals, several applets were developed for analysis, simulation, and visualization. These applets will aid in accurate prediction of potential failure modes as well as stress/strain limits on the flagpole. While some applets were designed for the flagpole specifically, many others were written strictly to collect and display data graphically. The latter applets are part of the overall goal to create a platform-independent software package that can be used for analysis and monitoring of other physical systems. Some of them can be used in other I-Campus projects to view the behavior of other physical systems.

The development of educational software is one of the main aspects of the I-Campus project. The goal is to improve several facets of structural engineering education, such as:

- Encouraging students to use software tools to solve engineering problems
- Empowering teachers with tools to measure how students learn most effectively

The educational applets developed for I-Campus provide a good opportunity for students to deepen their knowledge of structural engineering principles gained in lectures or start learning these principles on their own. They cover several areas of structural engineering, but the limited scope of the project and the restricted time resources certainly did not allow us to cover the complete range of this science. All applets developed during the project offer the user an opportunity to gain hands-on experience about possible abstractions of real world systems and about the influence of certain parameters on the results of a structural analysis. The underlying philosophy is to give the student as much influence on the setting of a structural system as possible in order to explore the effects of changing parameters on system behavior. Additionally, on-line help and tutorial sections are provided, which contain the theoretical background of the concepts. Little initial knowledge is required before starting to use the software, although it is probably not suitable for beginners.

2.6 Future Scope

Although the current system implemented in the Flagpole Instrumentation Project has accomplished many things, there is much to do to scale the system up to the actual structure, as well as other systems. Some of the issues that need to be addressed in further iterations of the project include portability, scalability, and expansion of educational software.

The portability of the system is very important in order to apply the system to physical structures. Currently, the system is quite portable, but exposed wires, delicate construction, wireline data transfer, and power requirements pose major problems. Much needs to be done in order to make the system stable and independent enough to operate

effectively in the elements. Another part of the portability of the system is wireless communication. Technologies such as WaveLAN and Bluetooth can be integrated into the current system after further investigation in their capabilities is established. Initial attempts with these technologies will be discussed in Chapter 3 – Data Collection.

The current monitoring system includes a prototype of the flagpole with a single degree of freedom. However, the actual flagpole has multiple degrees of freedom, which complicates the simulation and visualization process. Also, transferring the system to other structures such as bridges or buildings needs to be investigated in order to accomplish the goals set in the I-Campus initiative.

As explained above, many educational applets have been created in order to assist engineering students in conducting experiments. The next step is to broaden the base of concepts to teach. Also, more sophisticated visualizations such as finite-element models could be created to aid in the analysis of data received from physical structures.

3 Data Collection

In any real-time monitoring system, a data collection solution must be put in place. A complete solution consists of the right mixture of hardware and software components. In a nutshell, hardware components collect data from the flagpole and transfer it to servers for analysis and visualization with software.

3.1 Hardware

The hardware used in the flagpole instrumentation project consists of two main categories: sensors and the data acquisition system. Sensors measure data on the flagpole and the data acquisition system collects it from the sensors and transmits to the data collection software.

Currently, there are three types of sensors used in the project: accelerometers, strain gauges, and thermocouples. An accelerometer is placed at the top of the prototype to capture its accelerations, which are in turn converted into displacements. Strain gauges are placed at both sides of the prototype at its base to measure its strains. Since accelerations and strains change very often, they are sampled at 100Hz by the data acquisition system. The thermocouple is present to record any changes in temperature, which will have an effect on the strains and accelerations of the prototype. Temperature is a fairly static measurement, so high frequency sampling is not required. Therefore, the thermocouple is sampled every 16 seconds.

The data acquisition system is a hardware device that collects analog data from sensors and then transforms it into digital data. In our project, the FieldPoint module offered by National Instruments is used. The FieldPoint module is actually composed of many different modules. There is a voltage input module that reads from sensors that have voltage output, a network module to transmit the data, and a power cell to power the entire system. The FieldPoint module collects data from the sensors and sends it to a data acquisition server. The module that we are using is a 16-channel unit, collecting data

from sixteen sensors simultaneously. It “digitizes” the data as it is collected from the sensors. The FieldPoint module contains its own microprocessor and has an Ethernet interface that allows it to be connected to the Internet. Therefore, the FieldPoint module has its own IP address and can be connected to via the Internet. The FieldPoint module is shown below in Figure 3.1.

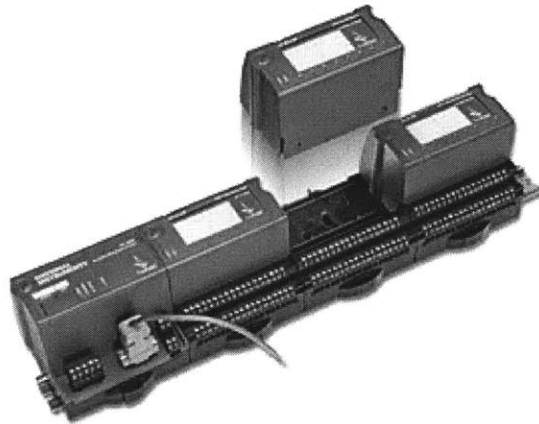


Figure 3.1 FieldPoint Module

The FieldPoint system features an innovative architecture that modularizes the communications, I/O functions, and signal termination. Therefore, the I/O, industrial network, and signal termination style can be independently chosen based on the particular application. Four classes of components exist that make this flexibility possible:

- **I/O Modules:** The FieldPoint system includes two general types of I/O modules. They are standard 8- and 16-channel modules and dual-channel modules for maximum mix-and-match flexibility. I/O modules provide isolated analog and digital inputs and outputs for a wide variety of signal and sensor types and are hot swappable and auto configurable for easy installation and maintenance. The FP-TB-10 module has been adapted for the project.
- **Thermocouple Modules:** This module is used for measuring millivolt signals from a thermocouple. Each channel of the module can be configured for one of a number of thermocouple types or millivolt ranges. The FP-TC-120 module has been adopted for the project.
- **Terminal Bases:** Thermocouple modules are installed on terminal bases that

provide terminals for field wiring connections, as well as module power and communications. The FP-TB-2 module has been adopted for the project.

- **Network Modules:** Network modules provide connectivity to open, industrial networks. The network modules communicate with the local I/O modules via the high-speed local bus formed by linked terminal bases. The FP-1600 Ethernet module and the FP-1000 serial module have been adopted for the project.

The FieldPoint module has proved to be very effective thus far in the project. However, there are still several challenges that we might have for this system in the foreseeable future, including:

- **Scalability:** Since each I/O module can only support a limited number of sensors, trouble could arise in the attempt to scale up to a larger structure when hundreds of sensors are to be deployed. A solution must be identified in order to handle large numbers of sensors with a minimum number of I/O modules.
- **Power Supply:** Providing a wireless solution for the communication of data is a primary concern. Therefore, it is not desirable to run a power cord to the modules either. A number of possible solutions have been investigated, such as lithium batteries, solar power, and fuel cells.
- **Durability/Security:** In order to protect the modules from theft or harm from the environment or vandalism, they must be placed in a safe location. Possible solutions include burying them or placing them in a protected area inside of a structure.

3.2 Data Communication

The next stage in the data collection process is the transfer of data from the FieldPoint module to the data acquisition server. Many different solutions were investigated and a number of technologies were explored in order to accomplish this task. The FieldPoint module has an Ethernet interface, so the immediate solution included running a network cable from the FieldPoint module to the network. This approach worked well to test the

connection from the FieldPoint module to the network, so all of the initial data was sent via this connection.

Since the FieldPoint module will be in the field when the actual flagpole is wired up, it is important to be able to access the FieldPoint module without running a cable to it. Therefore, getting to the FieldPoint module became a primary concern. Lantern batteries didn't even provide enough power, so a motorcycle or car battery would be the next size to try. Solar cells were also looked into to recharge the batteries, but the size of the solar panels would be far too large for practical use with the flagpole, for aesthetic reasons. Also, the Ethernet network module needs a lot of power to run. Powering the FieldPoint module is an important problem that is yet to be solved.

The data collected in its raw form has to go through a number of processes before useful information is extracted and made available for analysis. Hence, it is crucial to provide connectivity to the system that makes data acquisition and data flow unhindered and ensure that the entire process is efficient. The following are the criteria taken into consideration while designing the connectivity:

- High Bandwidth
- Minimum latency
- Minimum loss of data
- Easy maintenance
- Cost-effectiveness

In order to transmit the data wirelessly, a wireless technology that would allow large amounts of data to be transmitted over reasonable ranges was required. Two main technologies, WaveLAN by Lucent Technologies and Bluetooth by Ericsson were investigated.

WaveLAN is a long-range, broadband technology capable of transmitting 10-100Mbps over distances of up to 16 miles with an antenna. WaveLAN is a product from Lucent

Technologies used for wireless connectivity based on IEEE 802.11 standards. The WaveLAN solution for our system consists of two WaveLAN PC cards and an Orinoco RG-1000 (Residential Gateway). The WaveLAN PC cards are plugged into individual devices such as the FieldPoint module, laptops, and desktops, which communicate with each other. A residential gateway, RG-1000 acts as an access point and also a router. WaveLAN provides high bandwidth over long distances and is affordable.

Bluetooth is a low-power, short-range, broadband technology capable of speeds up to 1Mbps at distances of 10 meters currently (100 meters will be available in the near future). After inspecting both technologies, it was decided that Bluetooth would be most useful for the connection between the sensors and the FieldPoint module and WaveLAN for the connection between the FieldPoint module and the data acquisition server. However, Bluetooth is still in its infancy, so there is little support available for it. Therefore, the technology was not incorporated in the first iteration of the project, but preparations were made for its use in the near future.

In order to transmit data via a wireless connection, a hardware device was purchased to convert wireline to wireless data. An Ethernet and serial to wireless converter is installed on the FieldPoint module, as well as the data acquisition server. It should be noted that although the module works with both Ethernet and serial interfaces, it cannot convert one to the other, i.e., Ethernet wireless data is in a different format than serial wireless.

Upon further investigation, it became apparent that using the Ethernet interface included with the FieldPoint module limited the capacity of data sent across the network. Since Ethernet packets information, data was being lost when trying to send data faster than 8Hz. The problem is that FieldPoint module doesn't provide any buffering, so data must be read from the FieldPoint module as fast as it is read from the sensors. This problem removed Ethernet from the list of possible communication channels for the time being.

The next attempt included using the FieldPoint module's serial port to send data at higher frequencies. By transferring the data via a serial connection, data could now be sampled at much higher rates because of the lack of data packeting. However, with serial connections, networking is not possible, so the possibility of a sensor network was removed from this iteration. The next step was to use a wireless serial connection, but the wireless serial transmitter also packets data, resulting in the same problem as with Ethernet. Therefore, the current connection is a serial wireline connection.

3.3 Software

A data acquisition server running LabWindows collects the data transmitted from the FieldPoint module. LabWindows is a data collection and visualization software package created by National Instruments to interface with the FieldPoint module. The LabWindows software makes use of the CVI (C for Virtual Instruments) programming language to collect and display data.

Once the data acquisition server receives the data, it writes it to a data socket so that other programs may have access to it. A data socket is similar to a normal socket, but it requires some explanation. A normal socket connection consists of a peer-to-peer data connection between computers. On the other hand, a data socket connection is a client-server relationship in that data is broadcasted on the server, allowing any number of clients to access it simultaneously.

On the data socket server, the data is buffered in order to keep the performance high. The process is illustrated in Figure 3.2. The values from the sensors are stored as double precision values, and there are three sensors being monitored, so the buffer consists of a 3x16 array of double precision numbers. Strains and accelerations are the highest frequency values, so they are written to the first and second row in the array, respectively. The temperature only needs to be recorded every sixteen seconds, so it doesn't need to use an entire row in the matrix. Therefore, in the third row, the starting time is stored in

the first element, the ending time is stored in the second element, and the temperature is stored in the third element. Hence, each buffer contains 160ms worth of data.

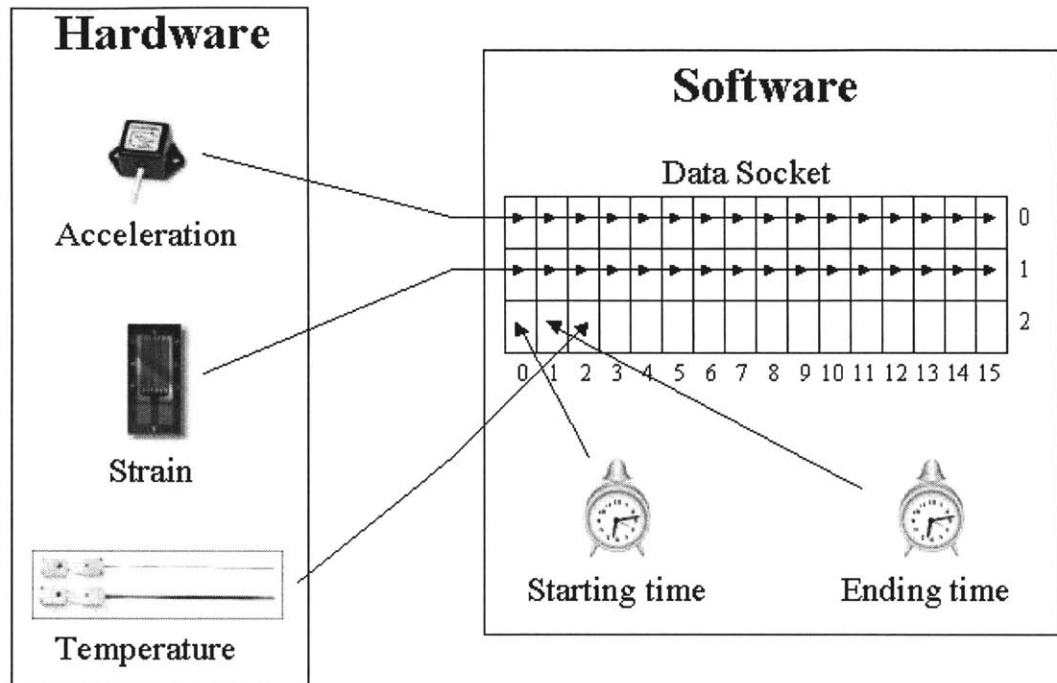


Figure 3.2 Data Socket

Since temperature readings are recorded every sixteen seconds, the same value is written to the matrix ten times in a row before a value is read from the sensor again. The times stored in the third row are represented as long values. However, they are stored as double precision values within the data socket. Initially, two arrays of data were written to on the data socket server – one 2x16 double array with the accelerations and strains and one float array to store temperatures. This approach has significant performance problems, but it is unclear why. The time is recorded when the first acceleration is read from the sensor and again when the last strain value is collected. Issues regarding the recorded time will be discussed next.

When the time should be recorded was a difficult issue to address. There are three main points in which the time could be recorded:

- As soon as it is sampled from the sensor in the FieldPoint module
- When the data is read from the data acquisition server
- When the data is written to sockets in the data socket server

Different trade-offs were experimented with to choose the best solution. For the purposes of the project, it is not necessarily important that the times recorded be completely accurate as to when they were sampled. However, the relative times are important, i.e., it is more important to know the spacing between samples than exactly what time it was recorded.

When trying to read the time directly from the FieldPoint module, the most accurate time available is to the second. Since the accelerometer and strain gauge are sampled at a rate of 100Hz, to the second accuracy is practically useless. The next step was to record the time within LabWindows. The accuracy of the Windows timestamp is to the millisecond, so that was not an issue. However, the actual times recorded were some cause for concern. In an ideal world, the timestamp for each sample should be exactly ten milliseconds apart, and the beginning and ending times written to the data socket should be exactly 160 milliseconds apart.

When actually implementing this, though, the times seemed to be somewhat sporadic – sometimes time values were repeated and other times, spacing was 20 milliseconds and larger. On the other hand, when one minute worth of data was analyzed, exactly 6000 samples existed. It is hypothesized that the reason for this is that the CPU is too busy switching between tasks to record the temperature at the exact intervals necessary. One possible solution is to only record the starting time for each sample set in the data socket.

The data type used for the time was also a cause for concern because of the many varied ways to represent a date used in this project, such as:

- Windows stores dates as a long value, representing the number of seconds since 1900. The number of milliseconds can also be retrieved with another system function
- Java represents dates as the number of milliseconds since 1970
- SQL Server has its own date data types

If the date is to be written to a data socket, it needs to be represented as a double, for performance reasons described earlier. The database can store the dates in almost any format. However, the most efficient way to store dates is the format that needs the least number of bytes to store it. It was decided to use the Java representation of dates to facilitate client interface with them. Therefore, the way dates are written to the data socket by CVI is as follows:

- The system time in Windows is multiplied times one thousand
- The number of milliseconds is added to it
- The number of milliseconds from 1900 to 1970 is also added
- It is cast as a double and written to the data socket

When the date is received from the data socket, it is converted into a long representation by a simple cast. The data type in which to store the time in was also nontrivial. The problems and concerns are explained in the next chapter.

4 Data Archiving

4.1 Database Management Systems

In order to archive data and be able to retrieve it for later analysis, a database is needed. Many different database management systems were investigated, including Oracle, AMOS II, and SQL Server.

Of these choices, Oracle is probably the best suited for this project because it is the most powerful and can be optimized and customized to suit our needs perfectly. However, along with the power of Oracle comes its complexity. Knowing that virtually no Oracle experience existed within the team and there was only 9 months to complete the project, it seemed as though it was too time-consuming to use Oracle for this iteration of the project. In hindsight, though, I feel that the project would have been better off had Oracle been adopted.

AMOS II is a main-memory, object-oriented database management system that allows data to be stored and retrieved very efficiently. However, the current system does not require the storage of objects. Also, persistent storage is more of an issue than quick access to the data.

SQL Server is Microsoft's high-end database management system. It has an easy-to-use graphical user interface to manage data. SQL Server was chosen because it is easy to learn and use, an MIT site license existed, and a feeling of obligation to Microsoft existed, since they funded the project.

The data model for our project is very simple; it is illustrated in Figure 4.1. Only two tables are needed to store the raw data, one for each frequency received from the sensors. There are also two summary tables, which will be explained in detail later.

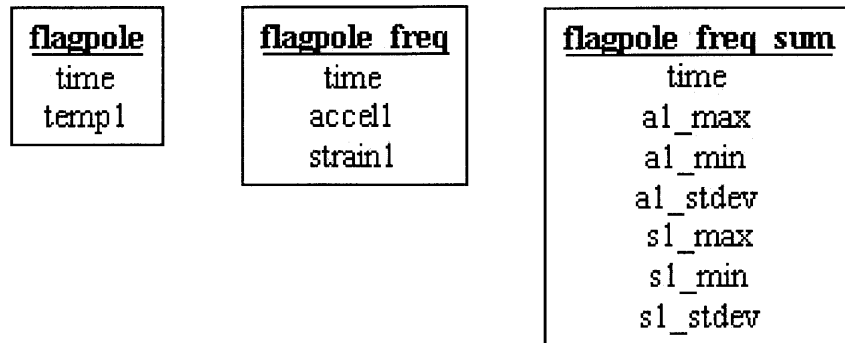


Figure 4.1 Data Model

As mentioned at the end of Chapter 3, the data type of the time variable was some cause for concern. Several choices were available:

- **datetime:** SQL Server date format. This is a good solution if many queries are to be run within SQL Server, because of its ease of use. However, the datetime data type is only accurate to 3 milliseconds, which poses a small problem when attempting to store highly accurate data.
- **smalldatetime:** another SQL server date format. It is only accurate to the minute, so it was immediately removed as a possibility for use in the project.
- **long:** any of the long representations described above could be used. Using a long representation makes ad hoc queries in SQL Server more difficult, but since most database interaction is through Java, Java's long representation proved to be the best solution to suit the needs of the project.

In either case, care had to be taken when defining indexes within the tables. A unique index could not be created because problems sampling the time discussed earlier caused multiple samples per millisecond. Therefore, the primary key was not made to be unique, and the performance of the database dropped substantially. The performance was so bad, it took nearly a minute to query and receive ten seconds worth of accelerations. Because of this, different archiving mechanisms were tested, such as:

- **Buffering in memory:** 24 hours of data is stored in a static array in order to keep the data in memory for quick retrieval. The biggest problem with this approach is

that there is no persistent storage of data. Therefore, if the archive program had to be restarted, all of the data stored in the variable would be lost. The archive program needed to run continuously for 24 hours to fill the arrays. At this point, another look was taken into AMOS II because of its main-memory capabilities.

- Reading directly from text files: since all of the data was being stored in text files on disk (explained in the next section), it made sense to use the text files as our persistent storage mechanism. Also, most queries on the data are simply a sequential search to find the first value, and then check against the ending value to end the search. One of the largest drawbacks of this method is that it is much more difficult to extract meaningful information from text files than from tables in a database, such as summary information or data mining techniques.

Neither of these solutions seemed to be the answer, so another look was taken at SQL Server. The next step was going to be to constrain the tables by making the primary key unique. Then I ran across clustered indexing, which I had used before, but not completely understood. The clustered index in a table is the order in which the data is physically stored on disk. Therefore, there can be only one clustered index per table. It should be noted that a clustered index does not need to be guaranteed to be unique. A clustered index is ideal for incremental searches and ranges in the primary key.

Range queries are executed in a very efficient manner when using a clustered index. An incremental search is performed on the starting date, and then each succeeding record is retrieved until the ending date is reached. Without a clustered index, an exhaustive search on the time field would have to be performed in order to achieve the same results. Therefore, range searches became extremely fast once the index was added. After adding a clustered index on the time field in the data tables, the time to run a query on ten seconds of accelerations decreased from almost a minute to less than a second.

In order for client applications to interact with the database, they must make a connection to the database. Once again, there are many possibilities that could be used, including

ODBC and JDBC. ODBC is the most general, as well as the most popular, because it is well defined and standardized. Java programs can connect to a database directly by using JDBC, or to an ODBC data source with a JDBC-ODBC bridge. Using a bridge is probably the slowest method because of the addition of another tier to the architecture.

Since many of the visualization tools were going to be written in Java, a JDBC driver was adopted for database access. When searching for a suitable JDBC driver, two major candidates stood out: FreeTDS and aveConnect. FreeTDS is a free, open-source JDBC driver that supports the TDS (Tabular Data Stream) protocol used by Sybase and Microsoft database management systems. We attempted to use FreeTDS for our project, but could not get the driver to work. The next attempt was to use aveConnect, a JDBC driver provided by Atinav, Inc. The aveConnect driver initially worked well for the purposes of the project. However, the newest version does not work with our current setup, so its use was terminated. The third and final driver used in the project is the JDBC-ODBC bridge that comes with the Java SDK. It works fine for the project; the only quirk is that an ODBC data source has to be loaded on every machine that uses this driver.

There are two ways for clients to gain access to a database – either the client downloads a JDBC driver every time an applet is requested, or the database querying takes place on the server and merely transmits the results to the client. It was decided that it is best to run the queries on the server for efficiency and security reasons. Therefore, a wrapper class for database access was created. Inside of it, the driver is registered with the driver manager one time, and then applets and other programs that need a database connection call a static method to return a connection to the database. This was created in order to make it much easier to change database drivers (and database management systems for that matter) as well as much more concise to code. Also, connection pooling can easily be implemented within the wrapper class, completely invisible to the client. The code for the database class is listed in Appendix A.

4.2 Archiving Process

After choosing a database and appropriate driver for it, an archiving process was created. The first approach was to use ODBC to write data directly to the database with CVI. Since data is being sampled at such a high rate, executing a SQL statement for each new record is very inefficient. This approach took up nearly 100% of the CPU time on the data acquisition server, which is unacceptable.

The next attempt was to read all of the values from the data socket like any other client and upload them to the database in batches, thus removing the processing from the data acquisition server. Also, I found that writing to a text file took almost no processing time whatsoever. Therefore, the process was to listen to the data socket, write values to a text file, upload them into the database from the text file, compress the text file, and then delete it, as shown in Figure 4.2. The code to execute this system is included in Appendix B.

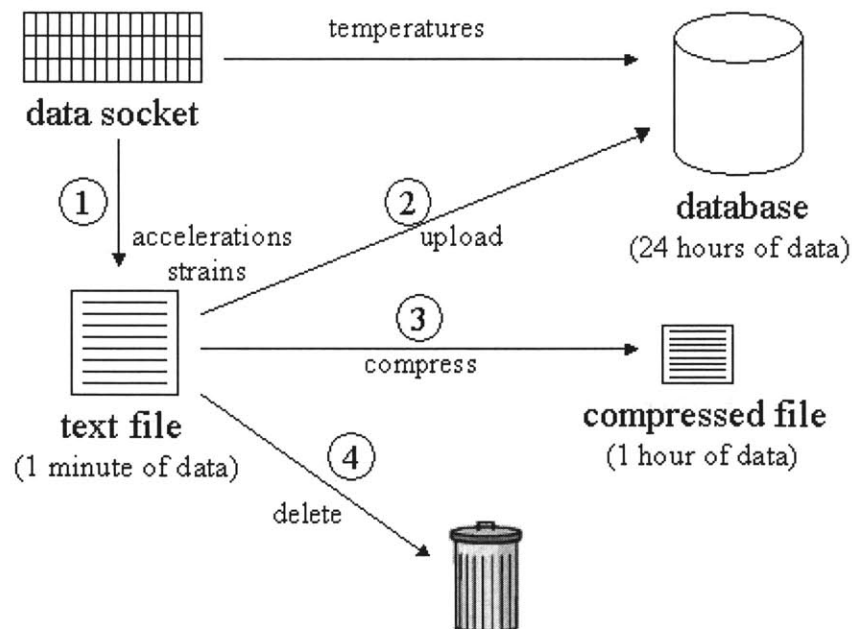


Figure 4.2 Data Storage

Buffering data in text files before uploading it to the database became the best solution for the project, but the implementation of this procedure raised even more questions and concerns. There is a tradeoff between the number of text files written, how recent the data in the database is, and how long it takes to upload the data into the database. For example, if data is written to a new text file every hour, there will only be 24 files on disk per day, but there will be a one hour “lag time” in the database, and it will take a significant amount of time to upload that many records into the database. On the other hand, if data is written to a new text file every minute, the files will upload quickly, the database will only have 1 minute of “lag time,” but there will be 1440 files on disk per day. Since it is more important to have recent information in the database and the speed at which the text files could be uploaded, a new text file every minute was created.

Not all information can be saved in the database for all time because so much data is being recorded. At the current rate of data collection, about 150 megabytes of data is added to the database every day if every sample is to be stored in the database. Therefore, a strategy was implemented in which only 24 hours of complete data exists in the database at any point in time. However, the size of the data can be significantly reduced while holding on to the essence of the data by summarizing it. A table in the database exists that contains summary information for the frequently recorded data. The summarization process is as follows:

- Collect all of the data for one minute
- Compute the minimum, maximum, average, and standard deviation for the acceleration and the strain within the range
- Write a new record to the summary table and remove the data from the original table

Using this method, there is a 1500 to 1 reduction in the amount of information. This information exists in the database for all time, totaling about 525,000 records per year, which is quite manageable in the database, while retaining much of the quality of the data. The first attempt to accomplish this summarization and purging task was to run a

maintenance task on SQL Server every hour that summarizes the data in the raw data table, inserts the records into the summary table, and then deletes all information in the raw data table that is older than 24 hours. However, if the archive program ever quit working, the maintenance task would continue to delete the old data and erase all of the data in the database after 24 hours. Therefore, the summarizing process was moved to the archive program and executed as a pair of SQL statements every hour, one to summarize and the other to purge.

Since so much data is being recorded, it is imperative that a compression technique be implemented in order to keep the server from running out of disk space. Compressing data within SQL Server was attempted, but it does not appear that SQL Server supports compressed data. Therefore, a compression technique was devised that made use of jar compression to compress the text files on disk. Another possibility and area of research is wavelet compression.

4.3 Compression Techniques

The first attempt at compressing data included compressing the text files using the jar utility in Java. Like everything else, this also had its tradeoffs. As mentioned above, 1440 text files are written to disk every day. In order to bring this number down and to compress the data, all of the text files were added to one jar file per day. This had performance problems because every new text file that was added to the jar file took longer and more processing time each instance. Therefore, only one hour's worth of text files exist in each jar file. This means that for every day, twenty-four jar files are created, each of which contains sixty text files.

The compressed data is only useful if it can be easily and efficiently reached. Therefore, a naming convention was developed in order to simplify compressed data retrieval. Jar files have the naming convention that includes the name of the table followed by the date and time it encompasses, accurate to the hour. For example, a jar file that contains information for the flagpole_freq table on February 12th, 2001 from 2:00-3:00pm would

be named `flagpole_freq2001-02-12-14.jar`. The text files have a similar format, but accurate to the minute.

Instead of buffering the data as lines in an ASCII text file and inserted into the database using the bulk insert command in SQL Server, the data could be buffered in a Java Statement object. Using this object, a batch of SQL statements can be executed at once by adding each to a Statement object and executed a batch at a time by using the bulk upload method. Using this method, a text file is no longer needed to buffer the data before it is inserted into the database. A problem with this method is that the data is not being compressed and stored on disk as it is with the current implementation. Therefore, a real-time compression technique was devised to write data directly to compressed files using wavelet compression. The process is described in detail next. If implemented correctly, this could serve as a better archiving solution.

When monitoring a physical structure such as the prototype in the Flagpole Instrumentation Project, extremely large amounts of data are collected continuously. Because of this, compression must be included to reduce the amount of data stored on disk. A wavelet analysis of the signal can split it up into several spectral components and the wavelet coefficients can be quantized based on the amount of information present in each channel. This will significantly decrease the size of the signal while retaining most of the information contained within it.

By applying a low-pass filter to a signal, the high frequency bands of the signal are removed and a smooth version of the original signal is obtained. A high-pass filter removes the low-frequency components of the signal, and the result is a signal containing the details (differences) of the original signal. By combining these two filters into a filter bank, the original signal is divided into an average signal and a difference signal. This can be applied recursively to the low- or high-pass channel.

For the purposes of this project, I have chosen to implement the Haar filter bank. The Haar filter is a two-tap filter with low-pass coefficients of $[\frac{1}{2}, \frac{1}{2}]$ and high-pass coefficients of $[\frac{1}{2}, -\frac{1}{2}]$. I chose the Haar because of its simplicity, straightforward implementation, and short delay. There are only two coefficients in the Haar filter bank, so 2^{levels} samples are needed to calculate all of the wavelet coefficients, where ‘levels’ represents the number of levels in the filter bank. For a three level filter bank, $2^3 = 8$ samples must be collected in order to create all of the coefficients. Because of the nature of wavelets and filter banks, only the lowest low-pass channel and all of the high-pass channels need to be recorded.

An analysis filter bank can be used to decompose the signal into its high and low frequency components by passing it through a combination of high- and low-pass filters. Most of the information will exist in the lower frequencies, while the higher frequencies are composed of mostly noise. Because of this, the wavelet coefficients for the high-pass channel can be represented with fewer bits than the corresponding low-pass channel. This process is recursively applied on the low-pass channel for the number of levels chosen. A three-channel analysis filter bank is illustrated in Figure 1 below.

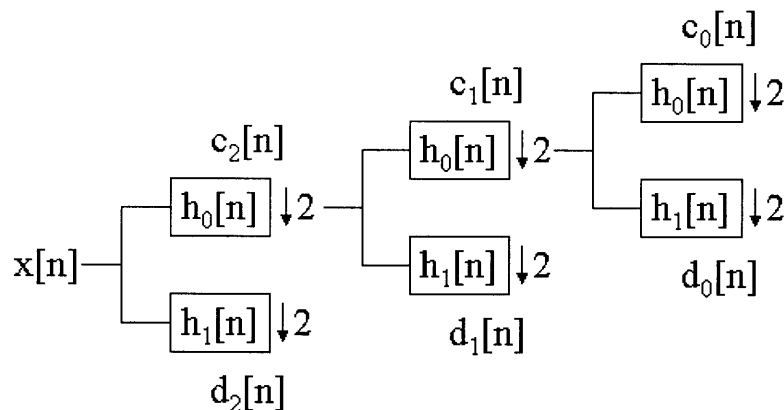


Figure 4.3 Analysis Filter Bank

The implementation of a 3-level filter bank like the one in Figure 4.3 requires seven arrays to store the data in memory: one for the original signal (x), three for the low-pass channels (c_2, c_1, c_0), and three for the high-pass channels (d_2, d_1, d_0). Also, a convolution

must be computed for each level, as well as a downsample. And, as mentioned above, eight samples are required to create all of the wavelet coefficients. The equations to compute these coefficients are shown in equations 4.1 and 4.2.

$$c[n] = (x[n] + x[n+1])/2 \quad (4.1)$$

$$d[n] = (x[n+1] - x[n])/2 \quad (4.2)$$

Using these equations, the original samples are split into high- and low-pass wavelet coefficients. Figure 4.4 shows the creation of each of the wavelet coefficient arrays during the decomposition of a signal passed through a 3-level Haar filter bank.

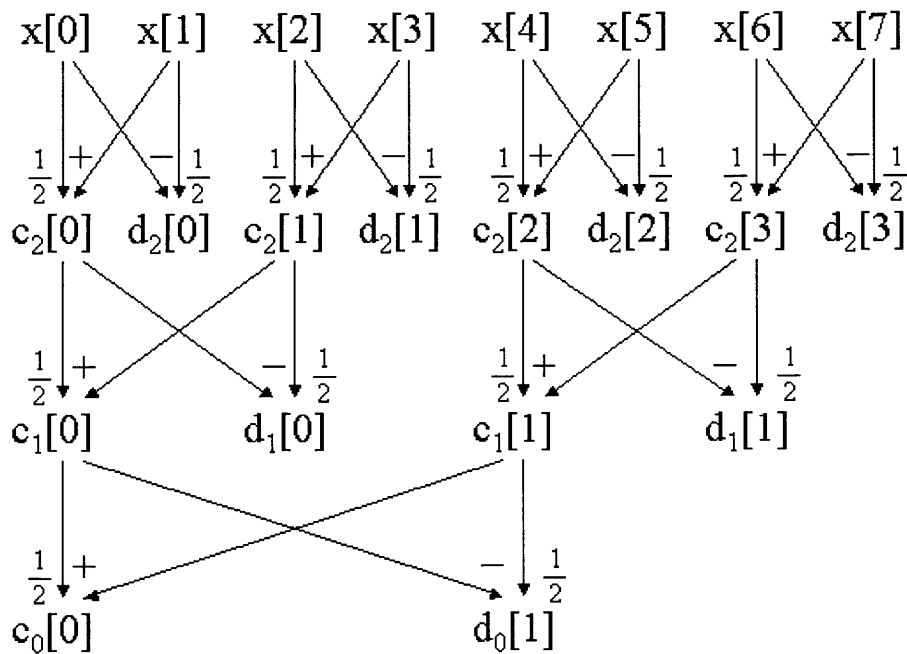


Figure 4.4 Haar Filter Bank Decomposition

This process can be greatly simplified and optimized by using a technique known as lifting. Using lifting, memory storage requirements and the number of calculations are significantly reduced. Calculations are performed in place in the original array, requiring no extra storage for the wavelet coefficients. The number of calculations is reduced

because there is no longer a need for the convolution and downsampling functions. The equations used to compute the coefficients are shown below in equations 4.3 and 4.4.

$$(d[n]) \quad x[n+1] = x[n+1] - x[n] \quad (4.3)$$

$$(c[n]) \quad x[n] = (x[n] + x[n+1]) / 2 \quad (4.4)$$

Using these equations, the filtering process is greatly simplified, and the transformation of the coefficients is illustrated in Figure 4.5 below.

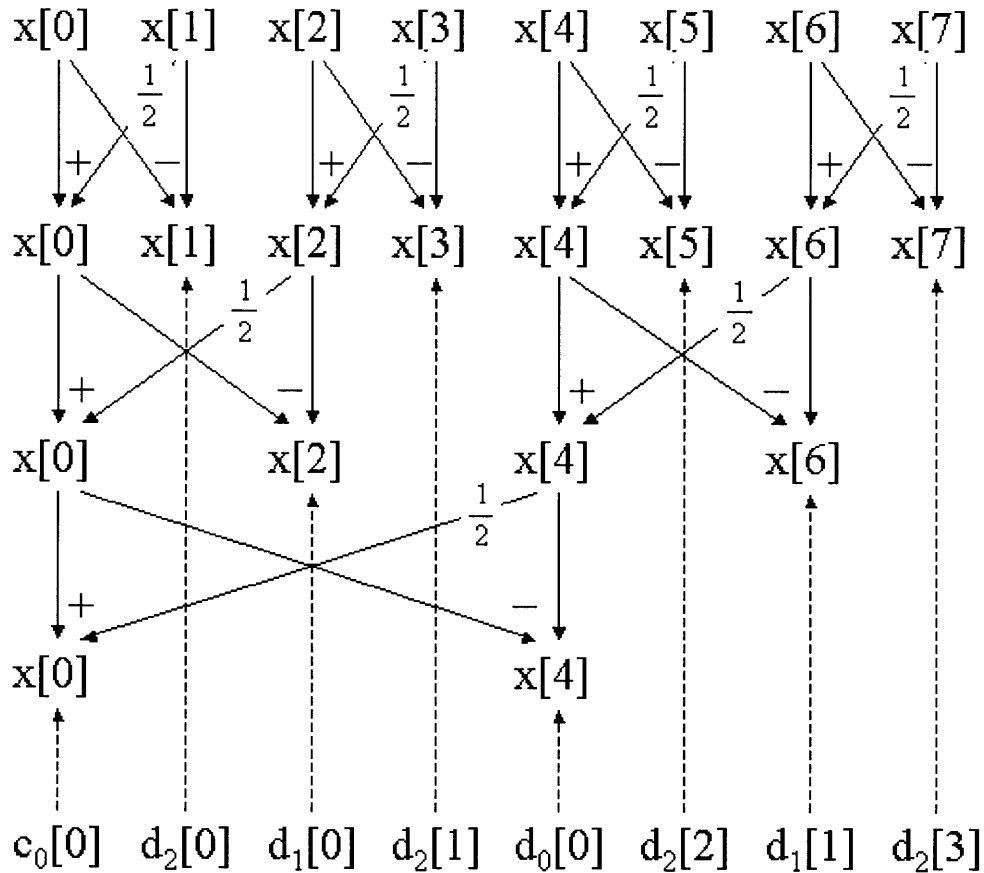


Figure 4.5 Haar Filter Bank Decomposition with Lifting

Once the signal has been decomposed into its frequency components, the coefficients are then quantized based on the amount of information contained in each channel. In order to quantize a coefficient with a given number of bits, first decide the maximum and

minimum expected values from the sample set and shift them so they begin at zero. For example, if values were only expected from -2 to 2 , the maximum would be set to 4 . Then, get the number in the range from 0 to $2^{\text{bits}}-1$, using Equation 4.5.

$$y = \frac{2^{\text{bits}} - 1}{4}x + \frac{2^{\text{bits}} - 1}{2} \quad (4.5)$$

Next, round to the nearest integer to preserve as much precision as possible. Finally, negative values are set to zero and values greater than or equal to four are set to four, thus keeping the data range from 0 to 4 . This is done to ensure that the data is between zero and four, even if the actual data range goes out of the boundary. The data range is stored as an integer to save storage space and to allow quicker calculations by performing bit shift operations instead of multiplications and divisions by powers of two.

After the data is quantized, it can be compressed by a variety of methods. For my project, I have chosen to make use of zip compression, using the `ZipOutputStream` utility available in Java. The `ZipOutputStream` class is very easy to use: simply create a new archive, add a new `ZipEntry`, and write to the compressed file. When writing to a `ZipEntry`, data is buffered in memory as a byte array, not actually written to disk until after the `ZipEntry` is closed. Once the `ZipEntry` is closed, compression techniques such as run-length encoding, Huffman encoding, and others are executed, compressing each `ZipEntry` as much as possible.

In order to reconstruct the signal, the coefficients must be expanded from their integer representation. The idea is basically the inverse of the quantization equation, and is illustrated below in Equation 4.6. After coefficient expansion, the values are again stored as double precision values and can be passed through the synthesis filter bank to recreate the signal.

$$y = \frac{4}{(2^{\text{bits}} - 1)x} - 2 \quad (4.6)$$

The purpose of the synthesis filter bank is to recreate the original signal from the wavelet coefficients. A three-level synthesis filter bank is shown below in Figure 4.6.

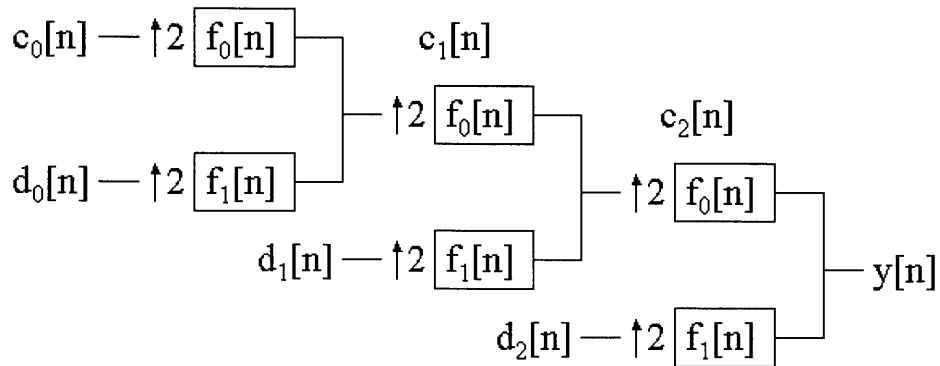


Figure 4.6 Synthesis Filter Bank

The actual implementation of the reconstruction of the Haar filter bank is very simple when lifting is incorporated. The process is to reverse equations 4.3 and 4.4, that is, change the order and flip the signs. Equations 4.7 and 4.8 show the reconstruction equations, and the process is shown pictorially in Figure 4.7.

$$(c[n]) \quad x[n] = (x[n] - x[n+1]) / 2 \quad (4.7)$$

$$(d[n]) \quad x[n+1] = x[n+1] + x[n] \quad (4.8)$$

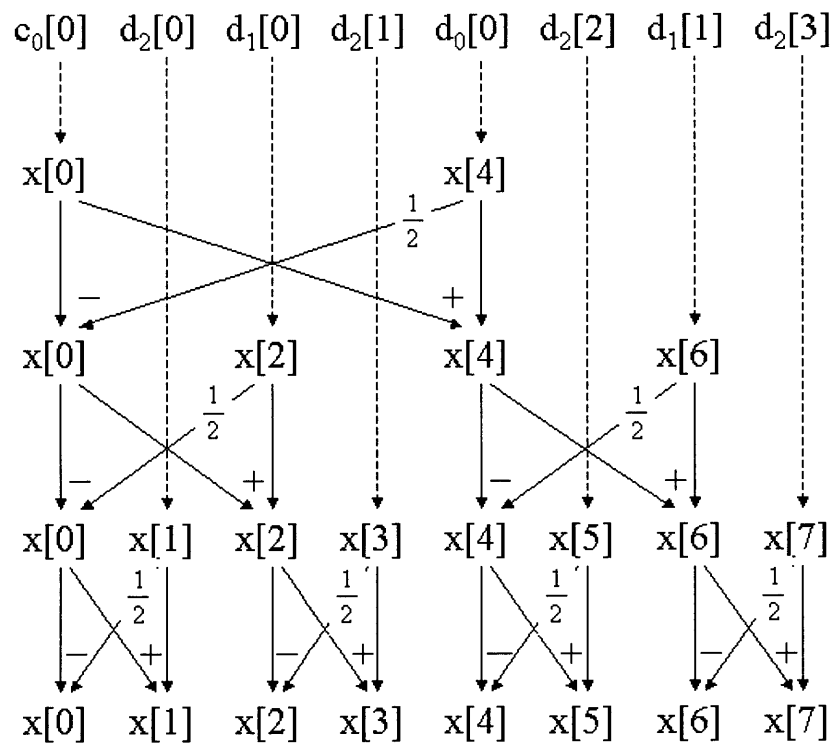


Figure 4.7 Haar Filter Bank Reconstruction with Lifting

Once the reconstruction is complete, the original signal will be created with a delay and any error introduced by the quantization of the high-pass channels.

To illustrate the process described throughout this section, I have created an applet that receives a signal from the prototype in the Flagpole Instrumentation Project, does an m-scale wavelet decomposition, compresses the coefficients, and displays a reconstruction based on the parameters chosen. A screenshot of the applet in action is shown below in Figure 4.8, and the code can be viewed in Appendix C.

The applet takes as a parameter an array of integers specifying the number of bits per level to store coefficients in, thus indicating the number of levels to traverse as well. For example, in the screenshot displayed in Figure 6, the parameter is [31, 16, 8, 8]. This is interpreted as creating a 3-level filter bank, using 31 bits to store c_0 , 16 for d_0 , 8 for d_1 , and 8 for d_2 . The applet is equipped to handle 2 to 6 levels. The individual coefficients

are displayed to show how well the filter is working and if more levels or less levels should be attempted. For example, if not enough levels are chosen, high frequencies show up in the low-pass channel, alerting the user that another level should be tried. On the other hand, if too many levels are chosen, less information will exist in the lowest channel.

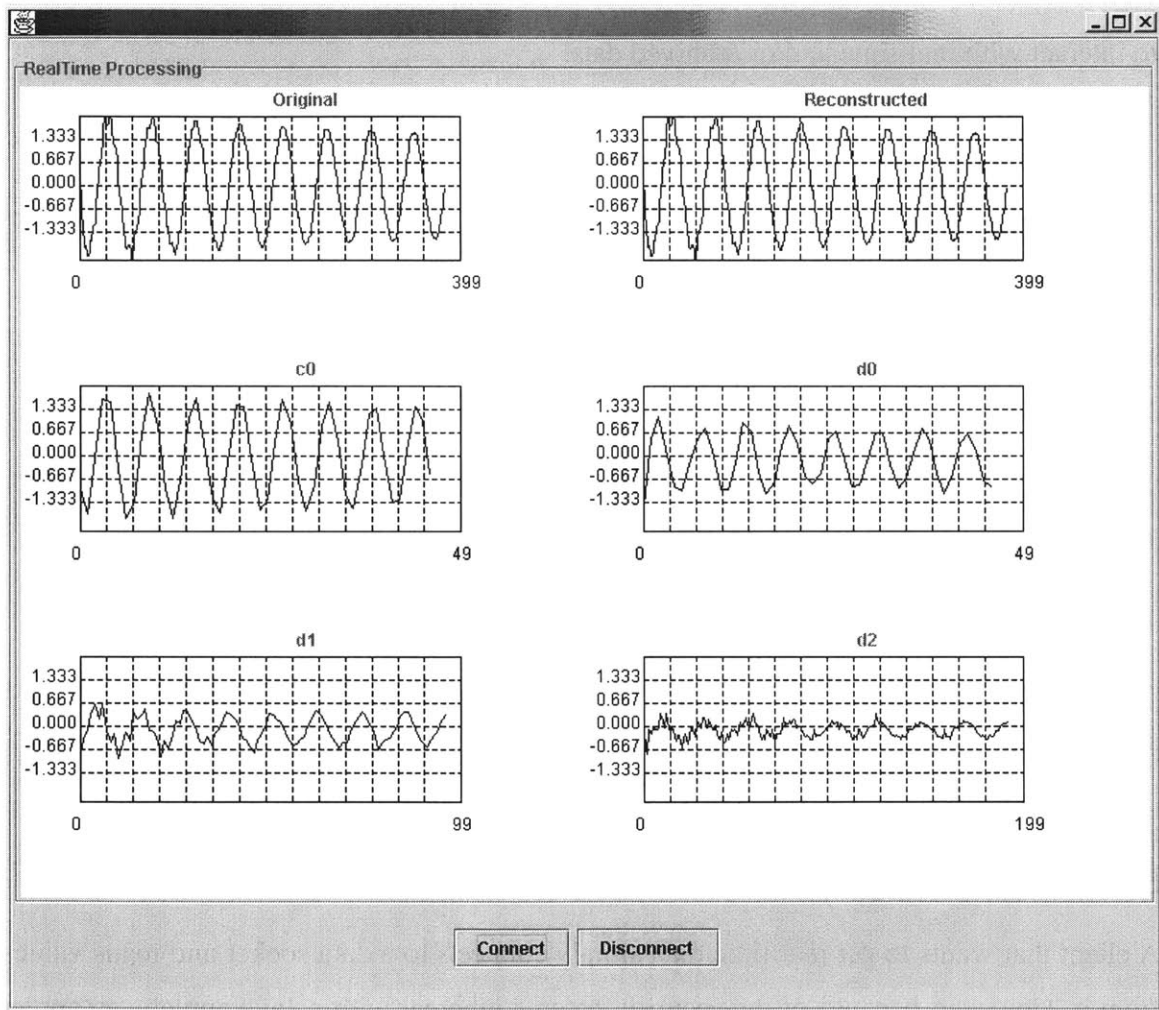


Figure 4.8 Wavelet Decomposition Applet

The reconstructed signal is displayed in the applet to illustrate how well the parameters were chosen. The number of bits per channel will determine how well the signal is reconstructed. If the parameters are chosen carefully, the reconstructed signal will look almost exactly the same as the original.

5 Client Interface

Although it is very important to collect and store the data, it must easily available to a client if it is to be useful. Client applications have the choice between real-time and two different types of archived data. Figure 5.1 shows the client interface overview used in the Flagpole Instrumentation Project. A set of APIs has been created in order for clients to interact with real-time and/or archived data.

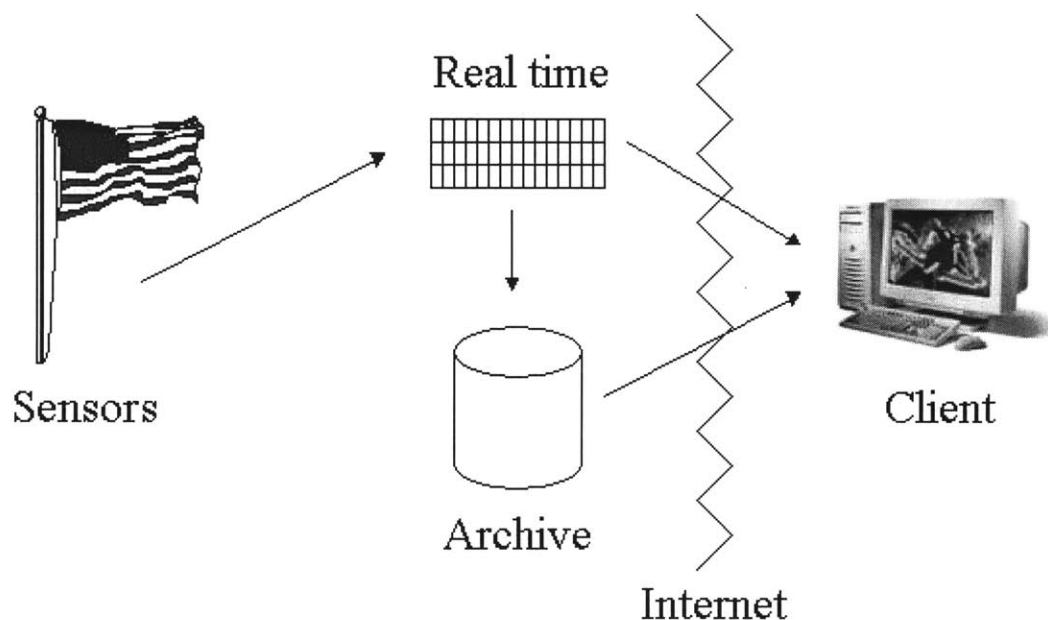


Figure 5.1 Client Interface Overview

5.1 Real-time Data

A client that wants to get real-time data simply connects to a data socket and reads values from it. However, because of the security features inherent within Java applets, an applet can only make a socket connection with the machine that it was downloaded from. Therefore, all applets that need to access real-time information must reside on the same physical machine as the data socket server.

Clients that need to make use of real-time data for visualization or analysis can connect to data sockets by using the `DataSocket` class provided by National Instruments. The

concept of data sockets was explained in Chapter 3. In order to listen to a data socket, a few steps must be taken.

- Create a new instance of a DataSocket
- Connect to a data socket URL
- Add a listener to the URL specified to update automatically when data changes
- Use the data on every update
- Disconnect from the data socket URL

Listing 5.1 shows an implementation of the DataSocket class:

```
DataSocket socket = new DataSocket;  
socket.connectTo("server/port", DSAccessModes.cwdsReadAutoUpdate);  
socket.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {  
    public void DSOnDataUpdate(DSOnDataUpdateEvent event) {  
        // do some processing  
    }  
});  
socket.disconnect();
```

Listing 5.1 Data Socket Code

5.2 Archived Data

Although it is important to retain all of the collected data in persistent storage, the data is of little use to client applications if it is not easily accessible. Some of the visualization applets created requires archived data for a specified purpose. There are two types of archived data that can be retrieved from the database:

1. One lump sum of information for analysis
2. Archived data that “behaves” like real-time data

In either case, as mentioned above, any information older than 24 hours exists only in compressed format. The complete system of retrieving archived data is illustrated in Figure 5.2. The client application requests archived data from the server. If any of the range is older than 24 hours, the necessary data is extracted from the compressed files

and uploaded to the database. This process will be explained in detail later. Once all of the requested data exists in the database, it is collected in a resultset. Depending upon the nature of the request, either the entire resultset is transferred to the client or it is buffered and written to an archived data socket that the client can then listen to.

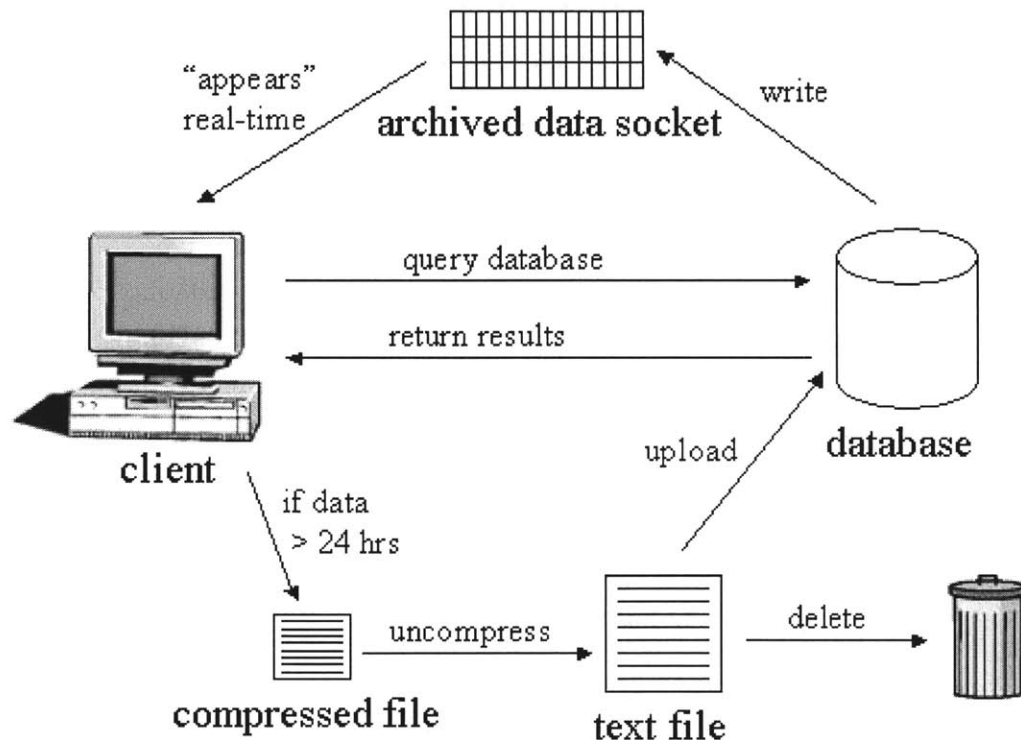


Figure 5.2 Archive Retrieval

For example, if you are watching a real-time visualization within an applet and you see something interesting happen at a particular point in time, you would like to be able to recreate that activity as if it was happening in real time. This is the situation portrayed in Figure 5.3. This applet was created to test whether an applet could display archived and real-time data within the same program. Users can view a graph of voltages from the accelerometer of the prototype in real time by typing the correct location into the Data Socket URL text box and clicking the "Connect" button. Archived data can be retrieved by entering the desired time into the text box in the lower right-hand corner pressing the

“Database” button. In either case, the applet can be disconnected from the data socket by pressing the “Disconnect” button. The code for this applet is included in Appendix D.

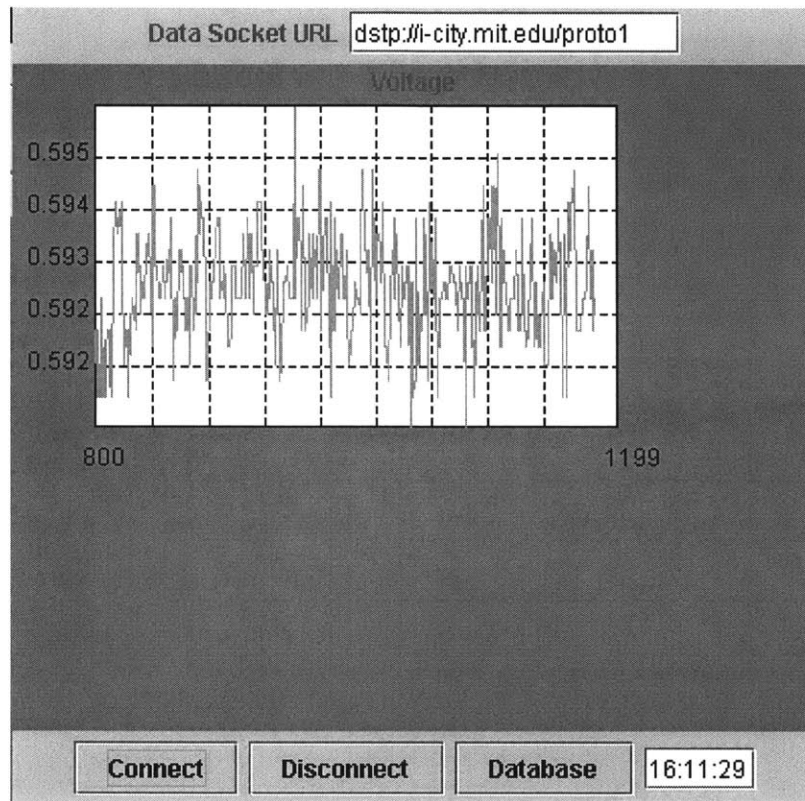


Figure 5.3 Archived Data “Behaving” Like Real-time

The other type of archived data available for client applications is querying the database and receiving a complete resultset in one large bundle. This type of archived data can be useful for many different applications. For example, a data mining tool that needs a range of data to extract pertinent information from the database and create a visualization from it.

Another possibility is the case in which an applet displays the variations in temperature over a 24-hour period. In this case, data is retrieved in one large bundle, not streamed as in the real-time case. This situation is illustrated in Figure 5.4 below in an applet created by Raghunathan Sudarshan. This applet has the capability to take a date range as parameters, retrieve data from the database, and provide visualization for the specified

date range. To run this applet, simply enter a starting and ending dates and times into the text boxes in the top frame and press the “Run Query” button.

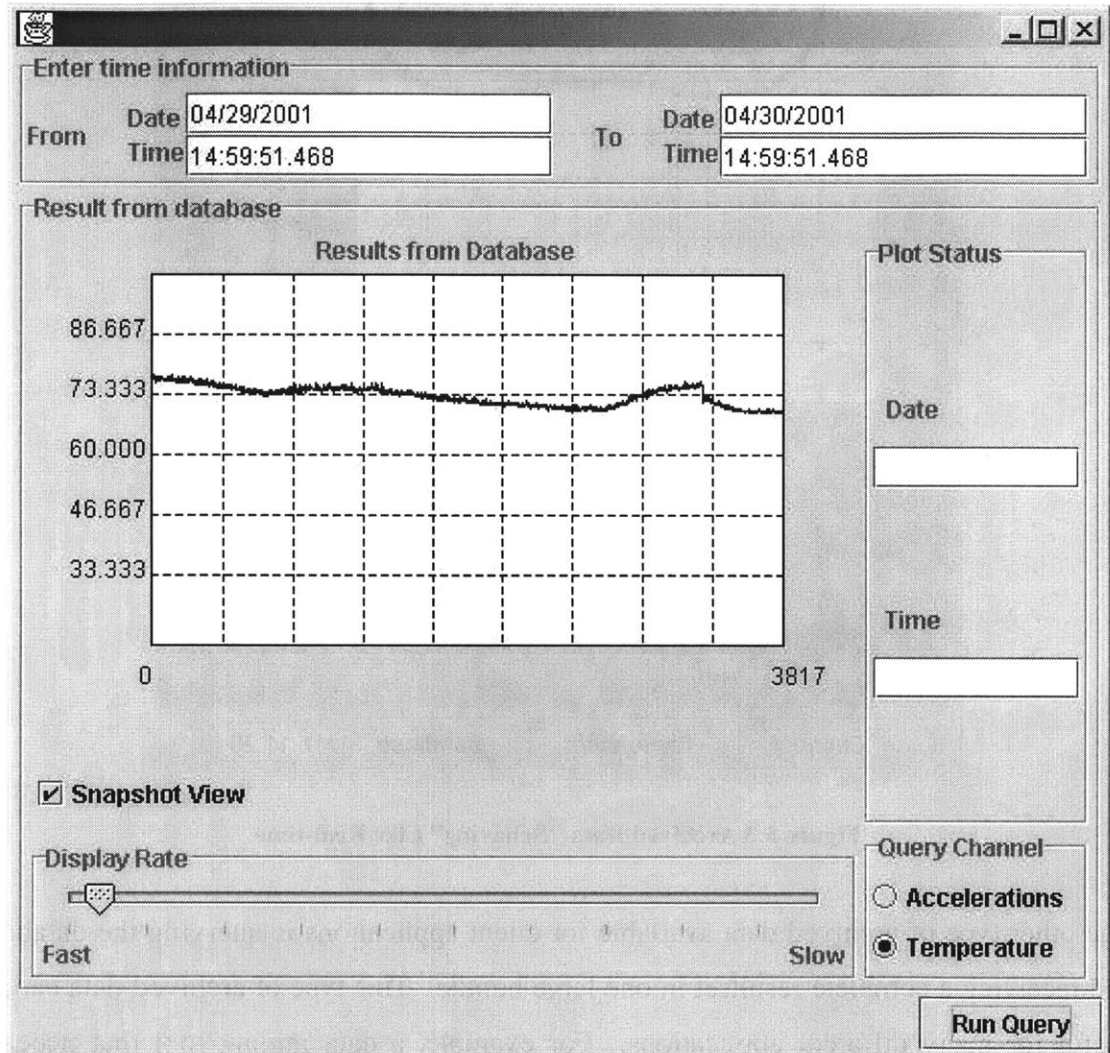


Figure 5.4 Archived Data Returned in One Bundle

Receiving data in one large bundle is straightforward – query the database and return a result set. Displaying data as if it were real-time is not quite so simple. There were two attempts to accomplish this task, either use servlets or RMI. Java Servlet technology provides web developers with a simple, consistent mechanism for extending the functionality of a web server and for accessing existing business systems. A servlet can almost be thought of as an applet that runs on the server side – without a face. Servlets

provide a component-based, platform-independent method for building web-based applications. Servlets have access to the entire family of Java APIs, as well as some HTTP-specific libraries.

The first approach to retrieve archived data and make it appear real-time was to use a servlet to execute stored procedures and write the results to a specified data socket. The basic procedure is as follows:

- An applet calls a servlet, passing at least two parameters through the URL:
 - Data socket to write to
 - Name of the stored procedure to execute
 - An optional list of parameters for the stored procedure
- Servlet parses parameters and executes stored procedure
- Servlet buffers data and writes to data socket in 16 value arrays every 160 milliseconds, mimicking real-time data socket

Because of the nature of this activity, a multithreaded solution needed to be implemented. If it ran on a single thread, the applet would call the servlet, the servlet would execute the stored procedure, write data to the data socket at regular intervals, then return control back to the applet. Since the applet needs to create a listener to the data socket it is reading from to track information, the listener would never be triggered because the sleep command in the servlet would freeze the entire thread.

Therefore, applets that required archived data were multithreaded in the first iteration. When archived data was requested, a new thread was spawned off that called the servlet, allowing the main thread to listen to the specified data socket. After analyzing the situation further, I found that the multithreading code could be written one time in the servlet, making the client much easier to code, while achieving the same results. The code for this servlet is included in Appendix E.

Remote Method Invocation (RMI) enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. Because many applets that require archived data also call for real-time data, the RMI server was placed on the same physical machine as the data acquisition server. The process works as follows:

- The client downloads an applet that requires archived data
- The applet calls an RMI method on the server to query the database
- The server executes the method and returns a value if one exists

Since RMI proved to be so useful to retrieve information from the database, it was adapted to distribute computing and storage between servers. There are two servers currently in use in the Flagpole Instrumentation Project – i-city and flagpole. I-city is the data acquisition and data socket server, while flagpole acts as the database and web server. Because of this physical separation, many problems arise regarding applet connection, storage, and processing. I-city is concerned with real-time data while flagpole deals with archived and static data. This system is illustrated below in Figure 5.5.

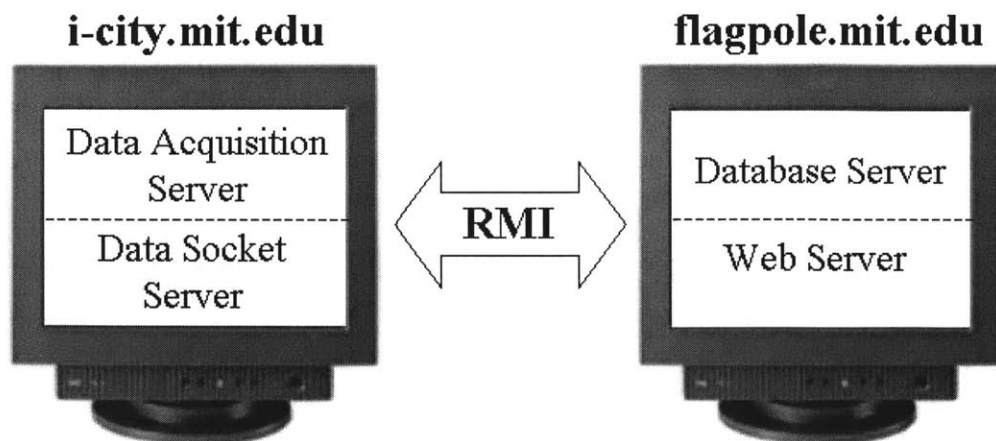


Figure 5.5 Distributed System

Since i-city deals with real-time data, it is important not to impair its performance. Therefore, only applications that deal with real-time data should run on i-city. It is better to run the programs and methods that read from and write to the text and compressed files, as well as database querying on flagpole. At first, almost all of the processing was done on i-city. As mentioned above, in order to allow applets to have access to real-time and archived data, the applets must reside on the same machine as the data socket server and the program that interacts with the database.

With RMI, a program can be run from one machine, but actually execute on another machine, only returning the results to the calling machine. Applets interface with i-city when they need data, from there RMI does the rest. If an applet requests real-time data, it is read directly from the data socket server on i-city. On the other hand, when an applet requests archived data, it must call an RMI method on flagpole to retrieve the information, thus using the processing power of flagpole to execute database queries while keeping the performance of i-city high. The RMI server and interface are included as Appendixes F and G, respectively.

As mentioned above, any information older than 24 hours exists only in compressed format. In order to read from these compressed files, a Java program called JarReader was created to extract this information from compressed format into the database. JarReader provides an interface for applets to automatically retrieve data from compressed files. Since the only time compressed data is required is when retrieving information from the database, the RMI server calls JarReader automatically when data is requested that is older than 24 hours. JarReader works as follows:

- Receives a starting and ending date as input parameters
- Collects a list of text files to be uploaded
- Extracts each text file from the jar file, uploads it, and then deletes it

This process is visualized above in Figure 5.2. It should be noted that all of this extraneous data would be deleted from the database as soon as the summarize-and-purge

batch program executes. JarReader is also an RMI server, relieving i-city of the processing time necessary to upload files. The code for the JarReader program is included in Appendix H, and its RMI interface is in Appendix I.

Because of the nature of the archive and JarReader programs – one writes to compressed files and the other reads from them, they were combined them into one RMI program. The portion of the code that archives the data still exists in the main method, but the section that reads from the compressed files is an RMI method. This is to enable the RMI server running on i-city to read from compressed files, using the CPU time on flagpole.

This made complete sense until it was actually implemented. An RMI server must be running on the remote machine before any programs that use its methods can be started. Since the RMI server running on i-city needs to connect to the RMI server on flagpole and vice-versa, an infinite loop is started. Hence, when the archive program and JarReader were combined, the program could never be started for the reasons stated above. Therefore, the programs were separated once again. This problem required us to start programs in a specific order. The order in which programs must be started is as follows:

- Close any of the programs running
- Start flagpole RMI server (JarReader)
- Start i-city RMI server (CVI Server)
- Start archive program on flagpole

6 Conclusions

After completing the first iteration of the Flagpole Instrumentation Project, I feel that a robust system has been established, and a lot of the details have been ironed out. I hope that this thesis gives some insight into the large amount of work put into the project and how it was an incredible learning experience for all involved.

The next iteration for the project includes scaling the current system up to include other structures and to apply the concept to other systems. Possible extensions include monitoring wells on the MIT campus to develop and test the concept of sensor networking. Another possibility is to monitor more mobile objects such as the MIT Safe Ride in real time to investigate GPS and wireless technologies. Pedestrian traffic in the hallways is another possible extension of the project. As one can see, although the initial focus of the Flagpole Instrumentation Project is to monitor and simulate physical systems, the concept can be extended to include other systems as well.

7 References

1. National Instruments Web Site. <<http://www.ni.com/fieldpoint/>>. (1 May 2001).
2. Strang, Gilbert, and Kevin Amaratunga. “Wavelets and Filter Banks.” 2001. Course 1.130/18.327. Massachusetts Institute of Technology.
3. Strang, Gilbert, and Truong Nguyen. Wavelets and Filter Banks. Wellesley, MA: Wellesley, 1997.
4. Sweldens, Wim, and Peter Schröder. “Building Your Own Wavelets at Home.” 1996. <<http://cm.bell-labs.com/who/wim/papers/athome.pdf>> (1 May 2001).
5. “The Java Tutorial.” 3 May 2001. <<http://java.sun.com/docs/books/tutorial/>> (1 May 2001).

Appendix A: Database Wrapper Class

```
import java.sql.*;

public class Database {
    public static Connection getConnection() throws SQLException {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (ClassNotFoundException e2) {
            System.out.println(e2.toString());
        }
        return
DriverManager.getConnection("jdbc:odbc:flagpole","flagpole","instant");
    }
}
```

Appendix B: Archive Program

```
import java.io.*;
import java.sql.*;
import java.text.*;
import natinst.msl.datasocket.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Archive implements Runnable {
    private static final String JAR = "C:\\program
files\\jdk1.3\\bin\\jar.exe";
    private static final String DIR = "C:\\Data\\"; // archive dir
    private static final String BUFFER = DIR + "flagpole_freq.buf";

    private static PrintWriter out; // writer for flagpole_freq.buf
    private static DataSocket socket; // data socket listening to
    private static CallableStatement call; // inserts into flagpole
    private static CallableStatement call_sum; // sum flagpole_freq
    private static CallableStatement call_purge; // purge data > 24 hrs

    private static Statement stmt; // bulk inserts into flagpole_freq

    private static int minute; // amount of time in text files
    private static int hour; // amount of time in jar files
    private static int counter; // counter for inserting temperatures

    private static String jarTime; // current jar file being loaded
    private static String textTime; // current text file being loaded
    private static String jarFile;
    private static String textFile;

    private static File file; // file object for all text files

    private static Connection con; // connection for flagpole
    private static Connection con_freq; // connection for flagpole_freq
    private static Connection con_purge; // connection for sum & purge

    private static SimpleDateFormat textFormat =
        new SimpleDateFormat("yyyy-MM-dd-HH-mm"); // text file format
    private static SimpleDateFormat jarFormat =
        new SimpleDateFormat("yyyy-MM-dd-HH"); // jar file name format

    public static void main(String args[]) throws IOException {
        try {
            con = Database.getConnection();
            con_freq = Database.getConnection();
            con_purge = Database.getConnection();

            // insert into flagpole table
            call = con.prepareStatement("{call spFlagpoleInsert (?,?)}");
```

```

        // bulk insert into flagpole_freq
        stmt = con_freq.createStatement();

        call_sum = con_purge.prepareCall("{call spFlagpoleFreqSum}");
        call_purge = con_purge.prepareCall("{call spFlagpoleFreqPurge
(?)})");
    } catch (SQLException e) {
        e.printStackTrace();
        con = con_freq = con_purge = null;
        System.exit(-1);
    }

    jarTime = jarFormat.format(new java.util.Date()); // get time
    hour = Calendar.getInstance().get(Calendar.HOUR); // get hour
    minute = Calendar.getInstance().get(Calendar.MINUTE); // get min
    out = new PrintWriter(new FileWriter(BUFFER), true); // buffer

    // listen to real time data socket and write info to database
    socket = new DataSocket();
    socket.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
        public void DSOnDataUpdate(DSOnDataUpdateEvent event) {
            writeData(event);
        }
    });
    socket.connectTo("dstp://i-city.mit.edu/protol",
DSAccessModes.cwdsReadAutoUpdate);
    System.out.println("Listening to data from dstp://i-
city.mit.edu/protol");
}

private static void writeData(DSOnDataUpdateEvent event) {
    DSData fData = socket.getData(); // get info from data socket
    double readData[][]; // array to hold values

    try {
        readData = fData.GetValueAsDoubleArray2D(); // retrieve data

        // upload files every minute
        if (Calendar.getInstance().get(Calendar.MINUTE) != minute) {
            minute = Calendar.getInstance().get(Calendar.MINUTE);
            out.close();

            // store name of text file created
            textTime = textFormat.format(new java.util.Date());
            textFile = DIR + "flagpole_freq" + textTime + ".tad";

            // insert upload date into file name
            new File(textFile).delete(); // delete upload file
            file = new File(BUFFER);
            file.renameTo(new File(textFile));

            out = new PrintWriter(new FileWriter(BUFFER), true);

```

```

        // start new thread to avoid data loss
        new Thread(new Archive()).start();
    }

    // insert record into flagpole every 16 seconds
    if (++counter == 100) {
        counter = 0;
        try {
            call.setDouble(1, readData[2][1]);
            call.setFloat(2, (float) readData[2][2]);
            call.executeUpdate();
        } catch (SQLException e2) {
            e2.printStackTrace();
            con = con_freq = con_purge = null;
            System.exit(-1);
        }
    }

    // write freq data to text file with evenly spaced times
    double interval = (readData[2][0] - readData[2][1]) / 15;
    for (int i = 0; i < 16; i++) {
        stmt.addBatch("INSERT INTO flagpole_freq values("
            + (long) (readData[2][0] + i * interval) + ","
            + readData[0][i] + ","
            + readData[1][i] + ")");

        out.println((long) (readData[2][0] + i * interval) + ","
            + readData[0][i] + ","
            + readData[1][i]);
    }
} catch (DSDDataException e3) {
    e3.printStackTrace();
    con = con_freq = con_purge = null;
    System.exit(-1);
} catch (IOException e4) {
    e4.printStackTrace();
    con = con_freq = con_purge = null;
    System.exit(-1);
} catch (SQLException e5){
    e5.printStackTrace();
    con = con_freq = con_purge = null;
    System.exit(-1);
}
}

public void run() {
    try {
        stmt.executeBatch();
    } catch (SQLException e5) {
        e5.printStackTrace();
        con = con_freq = con_purge = null;
        System.exit(-1);
    }
}

```



```

// create or add to jar files
String command[] = new String[4];
command[0] = JAR;
jarFile = DIR + "flagpole_freq" + jarTime + ".jar";

if (Calendar.getInstance().get(Calendar.HOUR) != hour || !new
File(jarFile).exists()) {
    hour = Calendar.getInstance().get(Calendar.HOUR);
    jarTime = jarFormat.format(new java.util.Date());
    jarFile = DIR + "flagpole_freq" + jarTime + ".jar";
    command[1] = "cf";

    Thread thread = new Thread(new Runnable() {
        public void run() {
            // summarize and purge freq data
            try {
                int total = call_sum.executeUpdate();
                System.out.println(total + " rows inserted");
                // purge data older than 24 hours
                call_purge.setLong(1, (long) (new
java.util.Date().getTime() - 86400000L));
                total = call_purge.executeUpdate();
                System.out.println(total + " rows deleted");
            } catch (SQLException e6) {
                e6.printStackTrace();
                con = con_freq = con_purge = null;
                System.exit(-1);
            }
        }
    });
    thread.start();

} else { // add to current jar file
    command[1] = "uf";
}
command[2] = jarFile;
command[3] = textFile;

// code that actually creates or adds to jar file
try {
    Process process = Runtime.getRuntime().exec(command);
    process.waitFor();
} catch (InterruptedException e7) {
    e7.printStackTrace();
} catch (IOException e8) {
    e8.printStackTrace();
}

// delete temp file
new File(textFile).delete();
}
}

```

Appendix C: Wavelet Decomposition Applet

```
import java.applet.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import natinst.msl.datasocket.*;
import java.util.*;
import java.io.*;
import PQ;
import java.util.zip.*;

public class Project extends JApplet {
    private int[] bits;
    private int levels;
    private int buffer_size;
    private DataSocket dsHandle;
    private StripChart s, recon;
    private StripChart[] charts;
    private JButton connect, disconnect;
    private ZipOutputStream zip;
    private ZipEntry entry;
    private PrintWriter out1;
    private final double magicNumber1=0.590/2.483;
    private final double magicNumber2=2.483;
    private final double magicNumber3=0.201;

    public Project(int[] bits_array) {
        bits = bits_array;
        levels = bits.length - 1;
        buffer_size = 1 << levels;
        charts = new StripChart[levels + 1];

        s = new StripChart(300,140,20,20,5,
StripChart.UpdateMode.block,400,"Original",1,true);
        recon = new StripChart(300,140,20,20,5,
StripChart.UpdateMode.block,400,"Reconstructed",1,true);
        charts[0] = new StripChart(300,140,20,20,5,
StripChart.UpdateMode.block,400/buffer_size,"c0",1,true);
        for (int i = 1; i < charts.length; i++) {
            int length = 400 / (1 << (bits.length - i));
            charts[i] = new StripChart(300,140,20,20,5,
StripChart.UpdateMode.block,length,"d" + (i-1),1,true);
        }

        connect = new JButton("Connect");
        disconnect = new JButton("Disconnect");

        try {
            out1 = new PrintWriter(new FileWriter("decode.txt"),true);
            zip = new ZipOutputStream(new FileOutputStream("data.zip"));
            entry = new ZipEntry("data.txt");
```

```

        zip.putNextEntry(entry);
    } catch (IOException e) {
        e.printStackTrace();
    }

    dsHandle = new DataSocket();

    formatGraphs();
    addListeners();
}

public void init() {
    JPanel chartPanel = createChartPanel();
    JPanel buttonPanel = createButtonPanel();
    Border etched = BorderFactory.createEtchedBorder();
    TitledBorder tb2 =
BorderFactory.createTitledBorder(etched, "RealTime Processing");
    chartPanel.setBorder(tb2);

    JComponent c = (JComponent)this.getContentPane();
    c.setBorder(etched);

    c.setLayout(new BorderLayout(c, BorderLayout.Y_AXIS));
    c.add(Box.createRigidArea(new Dimension(c.getWidth(),10)));
    c.add(chartPanel);
    c.add(Box.createRigidArea(new Dimension(c.getWidth(),10)));
    c.add(buttonPanel);
}

public void start() {}

public void stop() {
    dsHandle.disconnect();
}

private void plotData(DSOnDataUpdateEvent event) {
    DSData fData;
    double[][] readData;
    double[] rawData = new double[buffer_size];
    fData = dsHandle.getData();
    try {
        readData = fData.GetValueAsDoubleArray2D();
        int j = 0;
        for (int i = 0; i < 16; i++) {
            rawData[j++] = (readData[0][i] / magicNumber1 -
magicNumber2) / magicNumber3 + 0.03;
            if (j == buffer_size) {
                decomposeLevels(rawData);
                j = 0;
            }
        }
    } catch (DSDataException e1){
        System.out.println("Are you sure you are connected??");
    }
}
}

```

```

private void readFromFile() {
    Thread thread1 = new Thread(new Runnable() {
        public void run() {
            try {
                double[] rawData = new double[buffer_size];
                Reader r = new FileReader("newRaw.txt");
                BufferedReader bufferedReader=new BufferedReader(r);
                String nextLine;

                int i = 0;
                while ((nextLine=bufferedReader.readLine())!=null) {
                    Thread.sleep(10);
                    rawData[i++] = Double.parseDouble(nextLine);
                    if (i == buffer_size) {
                        decomposeLevels(rawData);
                        i = 0;
                    }
                }
                bufferedReader.close();
            } catch (FileNotFoundException e1) {
                e1.printStackTrace();
            } catch (InterruptedException e2) {
                e2.printStackTrace();
            } catch (IOException e3) {
                e3.printStackTrace();
            }
        }
    });
    thread1.start();
}

private void decomposeLevels(double[] x) {
    s.update(0,x,x.length);
    int[] q = new int[x.length];
    for (int i = 0; i < levels; i++) {
        int inc = 1 << i;
        for (int j = 0; j < x.length; j += (inc << 1)) {
            x[j + inc] -= x[j];
            x[j] += x[j + inc] / 2;
            q[j + inc] = quantize(x[j + inc], bits[levels - i]);
            charts[charts.length - i - 1].update(0, x[j + inc]);
        }
    }
    q[0] = quantize(x[0], bits[0]);
    charts[0].update(0, x[0]);

    x[0] = expand(q[0], bits[0]);
    for (int i = levels; i > 0; i--) {
        int inc = 1 << i;
        for (int j = (inc >>> 1); j < x.length; j += inc) {
            x[j] = expand(q[j], bits[levels - i + 1]);
        }
    }
}

```

```

    for (int i = levels - 1; i >= 0; i--) {
        int inc = 1 << i;
        for (int j = 0; j < x.length; j += (inc << 1)) {
            x[j] -= x[j + inc] / 2;
            x[j + inc] += x[j];
        }
    }

    recon.update(0,x,x.length);
}

int quantize(double x, int bit_length) {
    int max_value = (1 << bit_length) - 1;
    int y = (int)Math.round((max_value >>> 2)*x + (max_value >>> 1));
    if (y > max_value) y = max_value;
    if (y < 0) y = 0;
    return y;
}

double expand(int x, int bit_length) {
    return (4.0 / (double)((1 << bit_length) - 1)*(double) x - 2.0);
}

private void addListeners() {
    connect.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            readFromFile();
        }
    });

    disconnect.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                dsHandle.disconnect();
                zip.close();
            } catch (RuntimeException ex){
                System.out.println("Assertion Failed - Bailing out");
                System.exit(-1);
            } catch (IOException e2) {
                e2.printStackTrace();
            }
        }
    });

    dsHandle.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
        public void DSOnDataUpdate(DSOnDataUpdateEvent event) {
            plotData(event);
        }
    });
}

private JPanel createChartPanel() {
    JPanel stripChartPanel = new JPanel();
    stripChartPanel.setLayout(new GridLayout((charts.length + 1) / 2
+ 1, charts.length / 2));
}

```

```

stripChartPanel.add(s);
stripChartPanel.add(recon);
stripChartPanel.add(charts[0]);
for (int i = 1; i < charts.length; i++) {
    stripChartPanel.add(charts[i]);
}
return stripChartPanel;
}

private JPanel createButtonPanel() {
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new FlowLayout());
    buttonPanel.add(connect);
    buttonPanel.add(disconnect);
    return buttonPanel;
}

private void formatGraphs() {
    s.setAutoscale(false);
    recon.setAutoscale(false);
    s.setYBounds(-2.0,+2.0);
    recon.setYBounds(-2.0,+2.0);
    for (int i = 0; i < charts.length; i++) {
        charts[i].setAutoscale(false);
        charts[i].setYBounds(-2.0,+2.0);
    }
}

}

class ReaderMain {
    public static void main(String args[]) throws Exception {
        JFrame myFrame = new JFrame();
        myFrame.setSize(800,700);
        myFrame.getContentPane().setLayout(new BorderLayout());

        int[] bits_array = new int[args.length];
        for (int i = 0; i < bits_array.length; i++) {
            bits_array[i] = Integer.parseInt(args[i]);
        }
        Project wtDemo = new Project(bits_array);
        myFrame.getContentPane().add(wtDemo,"Center");
        wtDemo.init();
        wtDemo.start();
        myFrame.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        myFrame.setVisible(true);
    }
}
}

```

Appendix D: Real-time and Archive Data Visualization Applet

```
import java.applet.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import natinst.msl.datasocket.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.text.*;
import java.util.*;

public class RealTimeAndArchive extends JApplet {
    private DataSocket dsHandle;
    private StripChart stripChart;
    private JButton connect,disconnect,database;
    private JTextField urlField, timeField;
    private String dataSocketURL;
    private String someMessage = "Some Message";
    private ObjectInputStream in;
    private SimpleDateFormat formatter = new
SimpleDateFormat("HH:mm:ss");
    private SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd");
    private SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    public RealTimeAndArchive() {
        dataSocketURL = "dstp://i-city.mit.edu/proto1";
        urlField = new JTextField(15);
        urlField.setText(dataSocketURL);
        urlField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dataSocketURL = urlField.getText();
            }
        });
        stripChart = new
StripChart(300,200,20,20,5,StripChart.UpdateMode.block,400,"Voltage",2,
true);
        stripChart.setGraphColors(Color.gray,Color.black);
        stripChart.setGraphLineColors(0,Color.green);
        stripChart.setGraphLineColors(1,Color.red);

        connect = new JButton("Connect");
        connect.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dsHandle.connectTo(dataSocketURL,
DSAccessModes.cwdsReadAutoUpdate);
                someMessage = "Are you able to connect yet??";
                repaint();
            }
        });
    }
}
```

```

disconnect = new JButton("Disconnect");
disconnect.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        dsHandle.disconnect();
        someMessage = "Tried Disconnecting";
        database.setEnabled(true);
        repaint();
    }
});
database = new JButton("Database");
database.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        boolean ok = false;

        try {
            RMIInterface db = (RMIInterface)Naming.lookup("//I-
city/RMI");
            String startTime = sdf1.format(new Date()) + " " +
timeField.getText();
            Long time = new
Long(sdf2.parse(startTime).getTime());
            ok = db.writeToDataSocket("DSReader", "spDSReader",
new Object[]{time});
            Thread.sleep(160);
        } catch(Exception e1){
            e1.printStackTrace();
        }

        if (ok) {
            dsHandle.connectTo("dstp://i-city.mit.edu/DSReader",
DSAccessModes.cwdsReadAutoUpdate);
            database.setEnabled(false);
        }
        repaint();
    }
});
timeField = new JTextField(5);
timeField.setText(formatter.format(new Date()));

dsHandle = new DataSocket();
dsHandle.addDSOnDataUpdateListener(new DSOnDataUpdateListener() {
    public void DSOnDataUpdate(DSOnDataUpdateEvent e) {
        plotData(e);
    }
});

}

public void init() {
    JPanel buttonPanel = new JPanel();
    JLabel urlLabel = new JLabel("Data Socket URL");
    JPanel urlPanel = new JPanel();
    urlPanel.setLayout(new FlowLayout());
    urlPanel.add(urlLabel);
    urlPanel.add(urlField);
}

```



```

        buttonPanel.setLayout(new FlowLayout());
        buttonPanel.add(connect);
        buttonPanel.add(disconnect);
        buttonPanel.add(database);
        buttonPanel.add(timeField);
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(urlPanel,"North");
        getContentPane().add(stripChart,"Center");
        getContentPane().add(buttonPanel,"South");
    }

    public void start() {}

    public void stop() {
        dsHandle = null;
    }

    private void plotData(DSONDataUpdateEvent e) {
        DSData fData = null;
        double [][] readData;
        double [] oneChannel;
        fData = dsHandle.getData();

        try {
            readData = fData.GetValueAsDoubleArray2D();
            for(int i=0; i<16; i++){
                stripChart.update(0,readData[0][i]);
            }
        } catch(Exception e1) {
            try {
                oneChannel = fData.GetValueAsDoubleArray();
                for(int i=0;i<16;i++)
                    stripChart.update(0,oneChannel[i]);
            } catch (Exception e2) {}
        }
    }
}

class ReaderMain {
    public static void main(String args[]) {
        JFrame myFrame = new JFrame();
        myFrame.setSize(300,350);
        myFrame.getContentPane().setLayout(new BorderLayout());
        RealTimeAndArchive dsReader = new RealTimeAndArchive();
        myFrame.getContentPane().add(dsReader,"Center");
        dsReader.init();
        dsReader.start();
        myFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        myFrame.setVisible(true);
    }
}

```

Appendix E: Database to Data Socket Servlet

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import net.avenir.jdbc2.*;
import java.net.*;
import natinst.msl.datasocket.*;

public class DBtoDSServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        doGet(request, response);
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        String port = request.getParameter("port");
        String proc = request.getParameter("proc");
        String params[] = request.getParameterValues("params");

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        String title = "Test Servlet";
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<body>");

        try {
            DataSocket socket = new DataSocket();
            socket.connectTo("dstp://ecube.mit.edu/" + port,
                DSAccessModes.cwdsWriteAutoUpdate);

            String call = "{? = call " + proc + " ";
            if (params != null) {
                for (int i = 0; i < params.length - 1; i++) {
                    call += "(?, ";
                }
                call += "?)}";
            } else {
                call += "}";
            }
        }
    }
}
```

```

Connection con = Database.getConnection();
CallableStatement c = con.prepareCall(call);
c.registerOutParameter(1, java.sql.Types.CHAR);
ResultSet r = c.executeQuery();
double values[] = new double[16];
boolean done = false;

while (true) {
    for (int i = 0; i < 16; i++) {
        if (r.next()) {
            values[i] = r.getDouble(1);
        } else {
            done = true;
        }
    }
    if (done) break;
    socket.getData().SetValue(values);
    Thread.sleep(160);
}
socket.disconnect();
} catch (Exception e) {
    e.printStackTrace(out);
}

out.println("<h1>" + title + "</h1>");
out.println("</body>");
out.println("</html>");
}
}

```

Appendix F: RMI Server (i-city)

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import java.io.IOException;
import natinst.msl.datasocket.*;

public class RMIServer extends UnicastRemoteObject implements
RMIInterface {
    private static Registry reg;
    private Vector eventTimes;
    private static Connection con;
    private SQLException sqlEx;
    private DSDataException dsDataEx;
    private ArchiveInterface archiveInterface;

    public RMIServer() throws RemoteException, ClassNotFoundException,
SQLException {
        super();
        reg = LocateRegistry.createRegistry(1099);
        try{
            archiveInterface =
(ArchiveInterface)Naming.lookup("//flagpole/Archive");
        }catch(Exception e){
            System.out.println(e.getMessage()+" : Couln't start
RMIServer");

            System.exit(-1);
        }
        eventTimes = new Vector(100,50);
        System.out.println("Done with Constructor");
    }

    public Vector getHistoricalData(String proc, java.util.Date start,
java.util.Date end)
        throws RemoteException, SQLException, IOException,
NullPointerException {

        if (end.before(start)) {
            throw new NullPointerException("start date > end date");
        }

        if (proc.indexOf("freq") != -1) {
            java.util.Date minDate;
            Statement sql = con.createStatement();
            ResultSet rs1 = sql.executeQuery("select min(time) from
flagpole_freq");
            if (rs1.next()) {
                minDate = new java.util.Date(rs1.getLong(1));
            }
        }
    }
}
```

```

    } else {
        throw new NullPointerException("no info in database");
    }

    if (start.before(minDate)) {
        if (end.before(minDate)) {
            archiveInterface.extract(start, end);
        } else {
            archiveInterface.extract(start, minDate);
        }
    }
}

Vector eventData = new Vector(100, 50);
System.out.println("Got request");
eventTimes.removeAllElements();
String statement = "{call "+proc+"
("+start.getTime()+", "+end.getTime()+")}";
System.out.println(statement);
CallableStatement cs = con.prepareCall(statement);
ResultSet rs = cs.executeQuery();
while(rs.next()){
    eventTimes.add(new Long(rs.getLong(1)));
    eventData.add(new Double(rs.getDouble(2)));
}
return eventData;
}

public Vector getEventTimesAsLong() throws RemoteException,
NullPointerException
{
    if(eventTimes==null)
        throw new NullPointerException("eventTime is null");
    return eventTimes;
}

public Vector getEventTimesAsDate(SimpleDateFormat sdf) throws
RemoteException, NullPointerException
{
    if (eventTimes == null)
        throw new NullPointerException("eventTime is null");

    java.util.Date temp;
    Vector tempEvent = new Vector(eventTimes.size());
    for(int i=0;i<eventTimes.size();i++){
        temp = new
java.util.Date(((Long)eventTimes.elementAt(i)).longValue());
        tempEvent.add(sdf.format(temp));
    }
    System.gc();
    return tempEvent;
}

```

```

    public Vector get24HourTemp(java.util.Date today) throws
RemoteException, SQLException, IOException, NullPointerException
    {
        Vector v = new Vector(100,50);
        java.util.Date yesterday = new java.util.Date(today.getTime()-
864000001);
        v =
getHistoricalData("spGetHistoricalTemperatures",yesterday,today);
        return v;
    }

    public boolean writeToDataSocket(String port, String proc) throws
RemoteException, SQLException, DSDataException {
        return writeToDataSocket(port, proc, null);
    }

    public boolean writeToDataSocket(final String port, final String
proc, final Object params[]) throws RemoteException, SQLException,
DSDataException {
        final DataSocket socket = new DataSocket();
        socket.connectTo("dstp://i-city.mit.edu/" + port,
DSAccessModes.cwdsWriteAutoUpdate);

        String call = "{? = call " + proc + " ";
        if (params != null) {
            call += "?";
            for (int i = 1; i < params.length; i++) {
                call += ",?";
            }
            call += ")";
        }
        call += "}";

        CallableStatement c = con.prepareCall(call);
        c.registerOutParameter(1, java.sql.Types.CHAR);
        if (params != null) {
            for (int i = 0; i < params.length; i++) {
                if (params[i] instanceof String) {
                    c.setString(i + 2, (String) params[i]);
                } else if (params[i] instanceof Integer) {
                    c.setInt(i + 2, ((Integer) params[i]).intValue());
                } else if (params[i] instanceof Double) {
                    c.setDouble(i + 2, ((Double) params[i]).doubleValue());
                } else {
                    c.setObject(i + 2, params[i]);
                }
            }
        }

        final ResultSet r = c.executeQuery();
        if (!r.isBeforeFirst()) {
            socket.disconnect();
            return false;
        }
    }

```

```

Thread thread = new Thread(new Runnable() {
    public void run() {
        double values[] = new double[16];
        boolean done = false;
        while (true) {
            try {
                for (int i = 0; i < 16; i++) {
                    if (r.next()) {
                        values[i] = r.getDouble(1);
                    } else {
                        done = true;
                    }
                }
                if (done) break;
                socket.getData().SetValue(values);
                Thread.sleep(100);
            } catch (SQLException e1) {
                sqlEx = e1;
                break;
            } catch (InterruptedException e2) {
            } catch (DSDataException e3) {
                dsDataEx = e3;
                break;
            }
        }
        socket.disconnect();
    }
});
thread.start();
if (sqlEx != null) {
    throw sqlEx;
}
if (dsDataEx != null) {
    throw dsDataEx;
}
return true;
}

public long getServerTime() throws RemoteException
{
    java.util.Date serverTime = new java.util.Date();
    return (serverTime.getTime()-60000);
}

public void testDrive()
{
    System.out.println("Testing JDBC-Please wait...");
    try{
        java.util.Date d = new java.util.Date(getServerTime());
        System.out.println("Today is: "+d);
        Vector v = get24HourTemp(d);
        System.out.println("Size of queried items: "+v.size());
        if(v.size(>0)
            System.out.println("OK with query");
        else

```

```

        System.out.println("No records found over past 24 hours");
    }catch(Exception e){
        System.out.println("Caught exception: "+e.getMessage());
    }
}

public static void main(String args[])
{
    boolean aveConnect = false;
    if(args.length>0 && args[0].equals("aveConnect"))
        try{
            con = Database.getConnection();
        }catch(SQLException e){
            System.out.println("SQL error: "+e.getMessage());
        }
    else{
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con = DriverManager.getConnection("jdbc:odbc:flagpole");
        }catch(ClassNotFoundException e){
            System.out.println("Exception: "+e.getMessage());
        }catch(SQLException e){
            System.out.println("Exception: "+e.getMessage());
        }
    }

    //Cheat the security manager :-
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }

    String name = "//localhost/RMI";

    try {
        RMIInterface server = new RMIServer();
        RMIServer test = (RMIServer)server;
        test.testDrive();
        Naming.rebind(name, server);
        System.out.println("Ok Machees.Done Succesfully");
    }catch(Exception e) {
        System.out.println("Runtime Exception: "+e.getMessage());
        System.exit(-1);
    }
}
}

```


Appendix G: RMI Interface (i-city)

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Vector;
import java.text.SimpleDateFormat;
import java.sql.SQLException;
import java.io.IOException;
import natinst.msl.datasocket.*;

public interface RMIInterface extends Remote {
    /**
     ** Method to get any historical data in any time period
     **/
    Vector getHistoricalData(String proc, java.util.Date start,
        java.util.Date stop)
        throws RemoteException, SQLException, IOException,
        NullPointerException;

    /**
     ** Method to return the times at which events occurred
     ** [1] issue-what format should the date be returned??
     ** if the date is returned as long then the client will have to
     ** do the processing, but if the date is returned as a formatted
     ** date, will it be compatible with what the client wants??
     ** The solution is to pass in a java.text.DateFormat object
     ** that will allow the date to be formatted by the RMIserver.
     ** If no DateFormat is passed, a long is returned-much safer
     ** than returning a java.util.Date (hoffentlich)
     **/
    Vector getEventTimesAsLong() throws RemoteException,
        NullPointerException;

    Vector getEventTimesAsDate(SimpleDateFormat sdf) throws
        RemoteException, NullPointerException;

    /**
     ** Method to get 24hr temperature values
     **/
    Vector get24HourTemp(java.util.Date today) throws RemoteException,
        SQLException, IOException, NullPointerException;

    boolean writeToDataSocket(String name,String startTime,Object[]
        params) throws RemoteException, SQLException, DSDataException;

    boolean writeToDataSocket(String name,String startTime) throws
        RemoteException, SQLException, DSDataException;

    long getServerTime() throws RemoteException;
}
```

Appendix H: JarReader Program

```
import java.io.*;
import java.sql.*;
import java.text.*;
import natinst.msl.datasocket.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class JarReader extends UnicastRemoteObject implements
JarReaderInterface {
    private static final String JAR = "C:\\program
files\\jdk1.3\\bin\\jar.exe";
    private static final long MAX_DELTA_T = 10800000;
    private static final String DIR = "C:\\Data\\";
    private static SimpleDateFormat textFormat =
        new SimpleDateFormat("yyyy-MM-dd-HH-mm");
    private static SimpleDateFormat jarFormat =
        new SimpleDateFormat("yyyy-MM-dd-HH");

    public JarReader() throws RemoteException {
        super();
    }

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMI SecurityManager());
        }

        String name = "//flagpole/JarReader";
        try {
            Registry reg;
            if((reg = LocateRegistry.getRegistry(1099)) != null)
                reg = LocateRegistry.createRegistry(1099);
            JarReader jarServer = new JarReader();
            Naming.rebind(name, jarServer);
            System.out.println("Ready to accept connections");
        } catch (Exception e) {
            System.out.println("Runtime Exception: "+e.getMessage());
            System.exit(-1);
        }
    }

    public void extract(java.util.Date start, java.util.Date end)
throws RemoteException, SQLException, IOException {
        Connection con = Database.getConnection();
        Statement stmt = con.createStatement();
        con.setAutoCommit(false);

        StringBuffer uploadCommand = new StringBuffer("Insert into
flagpole_freq values(");
```

```

String command[] = new String[4];
command[0] = JAR;
command[1] = "xf"; // extract
long endLong = end.getTime(); // long representation of end date
boolean tooBig = false; // too much data requested flag

// set flag if too much data requested
if (start.before(new java.util.Date(endLong - MAX_DELTA_T))) {
    endLong = start.getTime() + MAX_DELTA_T;
    tooBig = true;
}

String textFileName;
File text_file;
Process process;

// loop from start to end date, iterate over minutes
for (long i = start.getTime(); i <= endLong; i+=60000) {
    command[2] = DIR + "flagpole_freq" + jarFormat.format(new
java.util.Date(i)) + ".jar";
    textFileName = DIR + "flagpole_freq" + textFormat.format(new
java.util.Date(i)) + ".tad";
    command[3] = textFileName;

    process = Runtime.getRuntime().exec(command);
    try {
        process.waitFor();
    } catch (InterruptedException e9) {
        e9.printStackTrace();
    }

    try{
        Reader reader = new FileReader(textFileName);
        BufferedReader bufferedReader=new BufferedReader(reader);
        String nextLine;
        while((nextLine = bufferedReader.readLine()) != null){
            StringTokenizer st = new StringTokenizer(nextLine);
            stmt.addBatch(uploadCommand+st.nextToken()+", "
+st.nextToken()+")");
        }
        stmt.executeBatch();

    }catch(FileNotFoundException e10){
        System.out.println("Could not find file: "+textFileName);
        continue;
    }

}

// throw exception if data range is too large
if (tooBig) {
    throw new IOException("date range too big");
}
}

```

Appendix I: JarReader RMI Interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.io.IOException;

public interface JarReaderInterface extends Remote {
    public void extract(java.util.Date start, java.util.Date end)
    throws RemoteException, SQLException, IOException;
}
```