Server Architecture for MEMS Characterization

by

Jared D. Cottrell

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

Bachelor of Science in Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1998

Author .....
Department of Electrical Engineering and Computer Science
August 24, 1998

Certified by .....
Donald E. Troxel
Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

Server Architecture for MEMS Characterization

by

Jared D. Cottrell

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Designers of Micro Electromechanical Systems need good tools to test the devices they fabricate. They need to be able to characterize a device's mechanical as well as electrical properties. Unfortunately, current tools provide no automated methods of analyzing the motion of a MEMS device. The research presented in this thesis defines and implements the server half of a remote MEMS Characterization System. It spells out the details of the system's network protocol for communicating between client and server. Finally, it describes how to deploy the server with other system components in a research environment.

Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering

Acknowledgments

There are countless people that deserve my thanks as I finish my graduate education here at MIT. I can only try to list them, knowing that I will miss many more than I can mention.

I would like to thank Professor Donald Troxel first for giving me the position as a research assistant that funded my graduate education. I am also very grateful for the experience and wisdom he shared with me through this past year. I would also like to thank Michael McIlrath for the many informal meetings and brainstorming sessions that helped focus my research.

Erik Pedersen has been a constant companion in the trenches. His advice and support has been a very welcome presence over the past year. I wish him luck in finishing up his own graduate work over the next semester. Yonald Chery has been a wonderful officemate. His war stories from the world of venture capital have been both entertaining and an excellent source of inspiration. It is largely because of Yonald that I found a position at Scient, itself a startup, where I will be signing on in September.

Much thanks goes out to the rest of my officemates in Building 36. William Moyne, Francis Doughty, Brian Lee, Matthew Verminski, and Thomas Lohman, have all contributed to the friendly and productive work environment. I come from a strong computer science background. It was a wonderful learning experience to work among so many practicing electrical engineers for a change. I think the insight I have gained will be invaluable in my career ahead.

I cannot neglect to mention all of the members of SPAMIT I have known and lived around through the years. I hope that many of them will prove to be life-long friends. A very heartfelt thank you goes out to all the people I have ever rowed with, whether through MIT Heavyweight Crew, Sloan Crew, or elsewhere. It is largely because of these teams that I have been able to survive five years at MIT.

Finally, an extra-big thank you goes out to my parents and my sister, Laura. Without their love and support for the past 22 years I would not have even made it to MIT in the first place.

This work was supported by the Defense Advanced Research Projects Agency (Air Force) under Contract F30602-97-2-0106.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this research is to provide a framework for building a MEMS Characterization System. It should also implement a prototype for a server conforming to this framework.

This chapter first gives an overview of the current state of the fields of MEMS and MEMS Characterization. It then discusses specific examples of previous works in the two areas. Finally, the chapter presents the organization of the thesis.

1.1 Micro Electromechanical Systems (MEMS)

MEMS are electronic devices, similar to microprocessors, except that they have mechanical properties too. MEMS are, in fact, designed and fabricated in much the same way that a microprocessor is. But new tools are needed to test MEMS devices. Test machines for microprocessors can only characterize the electrical properties of MEMS devices. Engineers also need to be able to analyze the mechanical properties of the MEMS devices they design. It is here that Computer Microvision comes into play as a tool to inspect MEMS devices for their mechanical properties.

1.2 Computer Microvision

Computer Microvision [9] is the use of a computer and a microscope for motion analysis. It uses common machine vision techniques to analyze microscopic objects. In Computer Microvision, we capture image data from a microscope and store it on a computer. We can then take advantage of the computer's raw processing power to characterize the object under the microscope.

1.3 MEMStation

To apply Computer Microvision techniques to the problem of testing MEMS devices, we will create a workstation we call a MEMStation. It will consist of a networked computer, a microscope, and some hardware to interface the two. To use all of this hardware, we will build a system we call the MEMS Characterization System[3, 15]. It will have a user interface that can work remotely, over the network.

More specifically, the hardware, apart from the computer and microscope, will include: a CCD camera, a frame grabber, an LED light source, a strobe pulse generator[18, 17], a stimulus signal generator, and a microscope controller. The camera and frame grabber will capture image data when told to and relay it to the computer. The stimulus signal generator will be controlled by the computer to stimulate the MEMS device under the microscope with the desired waveform. The strobe pulse generator will lock into the stimulus waveform and generate a pulse signal to turn on and off the LED light source during the appropriate phase of the stimulus, as dictated by the computer. Finally, the microscope controller will, when instructed by the computer, adjust microscope state variables such as focus, stage height, and magnification.

1.4 Related Work

1.4.1 Remote Microscope

A group of researchers at MIT have developed a Remote Microscope system[14, 16, 12]. Their system allows users to inspect microprocessors under magnification. That is, they can place a wafer under the system's microscope and take pictures of it.

The various clients that have been developed for the system allow the user to remotely control the microscope and related hardware. The server simply needs to have a computer, a microscope, some hardware to control the microscope from the computer, and a network connection.

The Remote Microscope system does not, however, have any way of stimulating a MEMS device. Nor does it have any way of strobing its illumination source to take the snap-shots necessary to analyze MEMS devices in motion. But even if the system did have some way of collecting the necessary data, it has no mechanism to analyze it.

1.4.2 Experiment Command Kernel (ECK)

The Experiment Command Kernel (ECK) is the current interface used by another group at MIT to capture still images of MEMS devices[10]. Unlike the Remote Microscope system, ECK allows the user to control hardware that can stimulate the MEMS device and strobe the illuminator. However, ECK must reside on the computer that is directly controlling the hardware.

ECK is essentially just a shell environment. By using various text commands the user can set a variety of variables related to capturing images. These variables include parameters for a stimulus waveform, parameters for a strobe illumination waveform, and various control parameters for the microscope.

ECK does not, however provide any mechanism for analyzing the images captured. The user must save the images to disk and analyze them with a separate program later. Also, ECK's user interface is less than ideal. As stated before, ECK must reside

on the computer that controls the microscope and other hardware. We would like to be able to have multiple users connected to a server simultaneously from remote computers. We would also prefer a graphical user interface to ECK's text-based command shell.

1.4.3 MEMS Analysis Tools

To process the data collected with ECK, the MIT research group has developed a set of MEMS Analysis tools. They can take the volumes of data that ECK has stored to disk and pass them through these programs. By doing so, they can discover various characteristics of MEMS devices under test.

One of these tools can detect sub-pixel motion in 3-D images[13]. It does this by watching for slight changes in grey-scale pixel values around the edge of an object. Such information is extremely useful in refining the accuracy of motion analysis.

Another tool can track a moving region of interest within a series of moving images[5]. The user might define an oscillating comb as the area of interest and track its motion through time.

While these tools are of great value in analyzing MEMS devices, they are not integrated with the image capture system, ECK. Like ECK, they rely on a text-based interface. Again, we would like to integrate these tools into a remotely controlled system with a graphical user interface.

1.5 Thesis Statement

Designers of Micro Electromechanical Systems need good tools to test the devices they fabricate. They need to be able to characterize a device's mechanical as well as electrical properties. Unfortunately, current tools provide no automated methods of analyzing the motion of a MEMS device. The research presented in this thesis defines and implements the server half of a remote MEMS Characterization System. It spells out the details of the system's network protocol for communicating between client and server. Finally, it describes how to deploy the server with other system

components in a research environment.

1.6 Organization of Thesis

This chapter has presented an overview of the background behind current MEMS characterization systems. First, it described the general problems to be addressed in analyzing MEMS devices. Then, it provided an overview of some current efforts to implement a MEMS Characterization System. Finally, it briefly outlined the implementation to be presented in this thesis.

The next chapter will provide an overview of the system architecture used in our effort to create a MEMS Characterization Server. Chapter 3 will detail the messaging protocol used to communicate between client and server in our system. Then, Chapters 4 through 7 will describe the four main modules in our server. The main text of the thesis concludes in Chapter 8 with some final thoughts on the system and possible future work.

We provide a detailed reference to the messaging protocol and all of the messages currently supported by our server. This reference document is included as Appendix A at the back of the thesis.

Chapter 2

System Architecture

2.1 Overview

One of our biggest concerns in designing the MEMS Characterization System was to provide a simple and well-defined client/server interface. With such an interface clearly spelled out and adhered to, we can build various different clients and servers that will all work with each other. For example, we may want to be able to test out a client implementation with a server that just simulates most system functions.

This chapter describes the overall system architecture of the MEMS Characterization Server. It outlines the basic functionality of each of the major system modules and how they interact.

2.2 Description

The Messaging Protocol is the glue that binds the entire MEMS Characterization System together. Again, it is the only interface through which the various clients and servers in the system can talk. The protocol is layered on top of the Hypertext Transfer Protocol (HTTP)[6]. HTTP is a well-understood protocol most commonly used in web browsers and web servers. As a result, we can take advantage of an off-the-shelf web server to both implement part of our Messaging Protocol and provide encryption and authentication routines for our server.

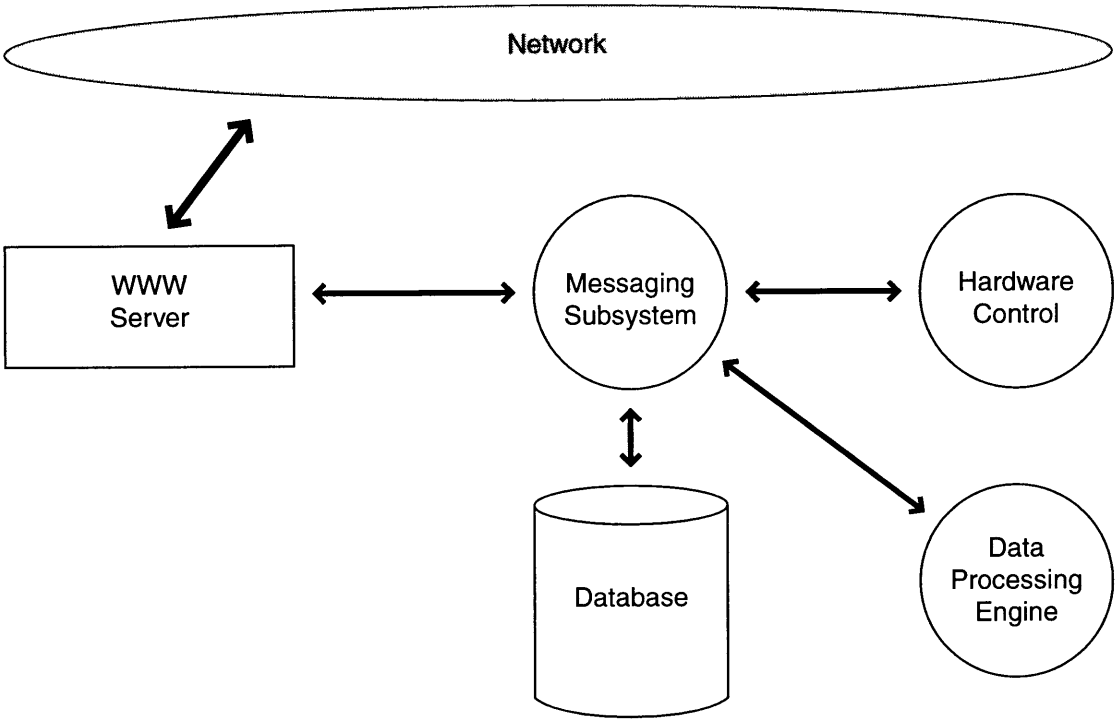

Figure 2-1: System architecture for MEMS Characterization System Server

The Messaging Subsystem sits directly behind the web server and implements our Messaging Protocol. It processes each incoming message and generates the appropriate reply message. When necessary, it passes off a request to another module for processing. This is the only module in the server that understands the Messaging Protocol. All internal communication between modules uses programmatic interfaces rather than messages.

The three remaining modules, the Database Subsystem, the Data Processing Engine, and the Hardware Control Module, handle specialized groups of functions. These functions directly correlate to groups of messages in the Messaging Protocol. When a request comes in to view or modify hardware state, the Hardware Control Module eventually handles it. A request for data, either storage or retrieval, invokes the Database Subsystem. And finally, a request to process data is sent to the Data Processing Engine.

2.3 Messaging Protocol

The well-defined Messaging Protocol used in our MEMS Characterization System allows for independent and parallel efforts to develop client and server applications. As long as the protocol is followed, a new client or server implementation should work with all existing client and server implementations.

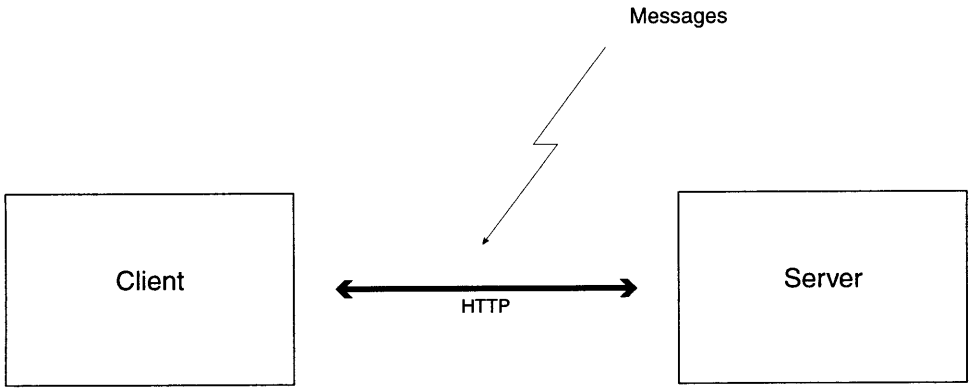

Figure 2-2: How the Messaging Protocol is used in the system

We decided early on that we wanted to use a web server as a gate-keeper to our system. We made this decision for many reasons. First, the design provides only one point of entry to the server. This means we have only one point of entry to make secure from unauthorized access. Second, the web server itself is a very well-understood technology. There are many off-the-shelf commercial web server implementations to choose from, so we do not have to implement one ourselves. Finally, because many available commercial web servers provide security services, we will not have to implement these services ourselves.

This decision to use a web server in our system greatly influences the design of our Messaging Protocol. Our protocol must be designed to be layered on top of HTTP. We accomplish this by sending text messages using the HTTP Post method and encoding them in the x-www-form-urlencoded MIME type. We choose to use messages in our interface because it provides a neat way of packaging and understanding discreet inter-module communications. The Post method is necessary to implement messages over HTTP because the alternative, called the Get method, limits the size of the text it sends to 256 characters. Post imposes no such limit. Messages are x-www-

form-urlencoded because that is the encoding scheme most commonly used with the Post method. We might therefore generate messages using other clients that send information with the Post method.

On top of our protocol, we define a set of messages that our system supports. These messages can be classified into functional groups. There are groups of messages to implement sessioning, inter-client messaging, database access, hardware control, and data processing, to name a few. In many cases, the modules in our server are designed to implement server functionality for one of these message groups.

2.4 Messaging Subsystem

In designing the server side of our system, we wanted to group all of the Messaging Protocol functionality into one module. That way, none of the other server modules need be modified should the protocol change.

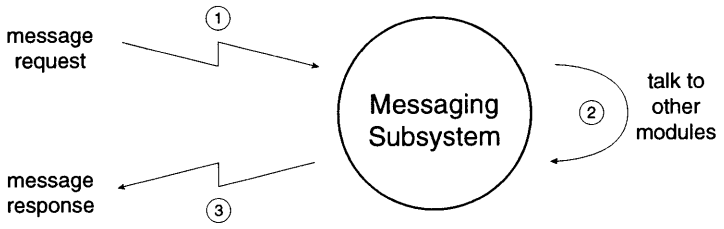

Figure 2-3: Interface overview for Messaging Subsystem

We call this module the Messaging Subsystem. It is linked to the web server and waits for requests from clients. As request messages come in, the module decodes the information and hands off control to one or more of the other server modules as necessary. The module also performs all error checking needed to enforce correct use of the Messaging Protocol.

To implement the functionality supported in the client/server interface, the Messaging Subsystem provides a message handler for each individual message. These handlers all export a standard programming interface. So writing a handler to support a new message type is as simple as copying and adapting the code for an existing handler.

Apart from the message handlers, the Messaging Subsystem also provides mechanisms to support specific features of the Messaging Protocol. To support sessioning, the module must keep track of all clients currently logged in. It must also provide an outgoing message queue to allow the server, as well as other clients, to send unsolicited messages to a client.

2.5 Database Subsystem

The Database Subsystem provides a central area to store all the data in the system. The functionality it exports reflects the functionality available in the database access message group of our Messaging Protocol.

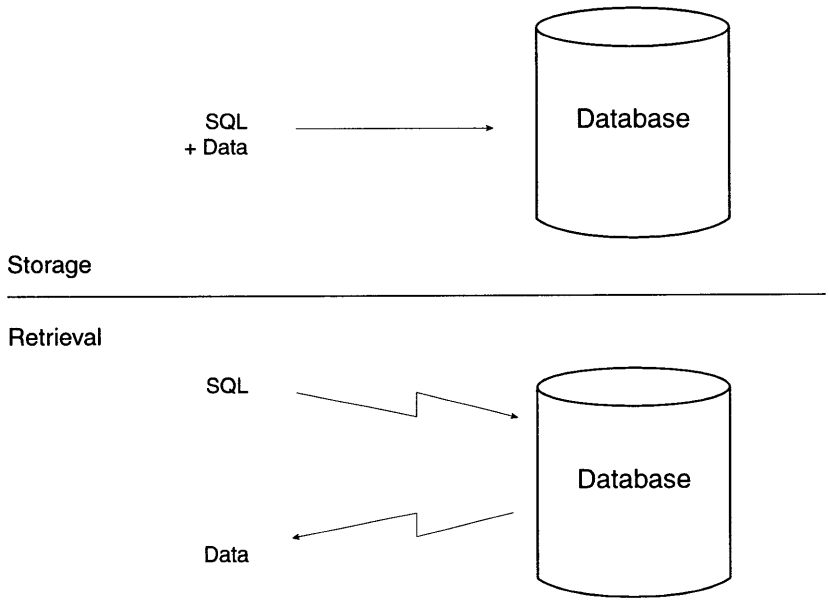

Figure 2-4: Interface overview for Database Subsystem

The Messaging Protocol defines two types of information in our system's database: data and meta-data. In our server implementation these two parts are stored separately. Meta-data is stored in a SQL database. The data itself, however, is stored as individual files in the file system.

This two part storage scheme is largely dictated by the structure of the Messaging Protocol itself. The protocol specifies that meta-data be accessed using messages over

the HTTP Post method. However, it also specifies that data be accessed through the more simple HTTP Get method. Our two part scheme lends itself directly to implementing the specifics of our Messaging Protocol. Messages requesting access to meta-data can be translated into SQL statements for the SQL database. Get requests, on the other hand, need no special handling by the server. The web server can simply respond by returning the file requested from the file system.

To connect to the meta-data in our SQL database, we use the Java Database Classes (JDBC). JDBC provides us a simple interface to access any SQL database we choose to install. For development purposes, we are using a mSQL server. But the system can be installed with nearly any major database engine.

Once again, our messaging protocol calls for data to be requested using the HTTP Get method. To get at the actual data in our database, a client need only request a URL that it finds in the meta-data database. Each piece of meta-data contains a reference, in the form of a URL, to the data it is associated with. Requesting that URL through the Get method returns the file associated with that piece of meta-data.

2.6 Hardware Control Module

The Hardware Control Module contains all of the hardware control functionality in the server. The hardware devices we need to control include a microscope, a frame grabber and camera, a strobe pulse generator, and a stimulus waveform generator. To implement most of this functionality, we need to access platform-specific device drivers. Such hardware device drivers are written entirely in native code, as is the code to access the drivers. As a result, most of this module is also written in native code.

We encapsulated all of the hardware control functionality in one module to be able to better manage our native code. We did not want to have native code in bits and pieces throughout several server modules. Instead, we have all of our native hardware control code in one place. Porting our server to a different hardware platform is therefore much simpler, as all the code to change is in one place.

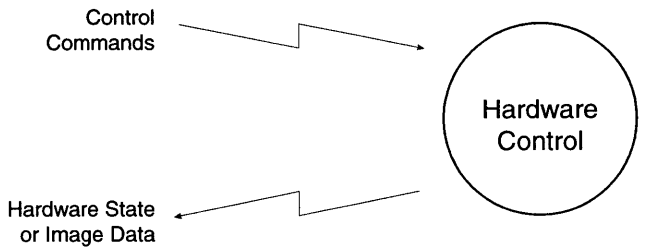

Figure 2-5: Interface overview for Hardware Control Module

To interact with our native code from our Java code, we use the Java Native Interface (JNI). JNI defines a way to write interfaces to native code from Java. Using tools that support JNI, we can write and compile native code implementations for our interfaces. In fact, the Java portion of this module is just a wrapper around an interface written using JNI.

Currently, this module is implemented only a stub that returns some basic default images when asked to capture things. It does not actually interface to any hardware. We use this implementation to simulate and test our server.

2.7 Data Processing Engine

The Data Processing Engine encapsulates our image and data analysis routines. A Data Processing Engine has one basic function: take in some data and some parameters and return some new data. A simple Data Processing Engine might take in a single image and no parameters. The data it returns might be the negative image of the original image. More complex engines might perform a three-dimensional analysis of a set of images.

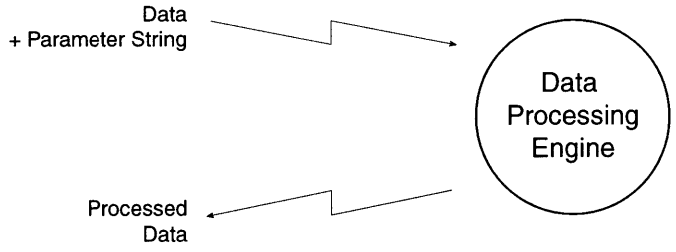

Figure 2-6: Interface overview for Data Processing Engine

The Data Processing Engine may not be included in all server implementations. Users may wish to perform their processing elsewhere in the system. For example, we might choose to copy images to a client that analyses them locally. As a result, in designing our system, we wanted to make the Data Processing Engine removable. We therefore chose to make it its own module.

Like the Hardware Control Module, the Data Processing Engine module also uses the JNI, but for a different reason. Processing routines are very CPU-intensive, so code optimization is a big concern. We have much greater control over the performance of the Data Processing Engine module if it is written mostly in native code. So, as in the Hardware Control Module, the Java portion of this module is just a wrapper around an interface written using JNI. And again, as with the Hardware Control Module, the Data Processing Engine is currently implemented as a simple stub for simulation purposes.

Chapter 3

Messaging Protocol

3.1 Overview

In this chapter we take a look at the interface between the client and server in our MEMS Characterization System. This interface is perhaps the most important aspect of the system. Once we understand it, we can spawn independent efforts to develop different clients and servers. When finished, they will all be able to interact with each other as if the teams had been collaborating all along.

3.2 Description

The Messaging Protocol, as we call it, is really made up of two interdependent parts. This first is the actual protocol that our interface uses. The protocol is a set of general rules that define how all messages are formatted and transmitted. Once we understand the underlying protocol, we can discuss the second part of our Messaging Protocol: the messages themselves. Our messages are divided into groups with common functionality. In many cases it is impossible to talk about one part of the Messaging Protocol without referring to the other. Many messages are essential to facilitating the protocol.

3.3 Protocol

The goal for our protocol design was to define a simple extension to some web protocol. By doing so, we can take advantage of existing web protocol implementations without having to reinvent the wheel. For example, our implementation uses a web server to implement the basic web services we need.

3.3.1 Protocol Basics

Our protocol is layered on top of the Hypertext Transfer Protocol (HTTP). HTTP is the basic protocol used by all web browsers and web servers to communicated between one another.

HTTP uses stateless connections. This means that the details of one connection does not affect another connection, as HTTP does not share information between the two. As we shall see, however, we can implement a protocol on top of HTTP that does use connections with state.

Each HTTP connection consists of exactly one request from a client and one response by the server. This clearly implies that all connections are initiated by the client. Servers using HTTP are only allowed to send information when it is explicitly request by a client.

HTTP messages, both requests and responses, are divided into two parts: headers and data. The headers section contains a set of fields describing the message. Each field appears on an individual line. Depending on the HTTP message being sent, some fields may be required. Most, however, are optional. The data section contains the guts of the actual message being sent. It is separated from the headers section by a single blank line. If a data section is included, its length, including return characters, must be specified in the Content-Length header field. Also, the method used to encode the data in the data section must be specified in the Content-Type field.

To make a request, a client can use one of two methods: Get or Post. The Get method is the more commonly-used of the two. It does not actually include a data

section. The Get method simply requests that the server return a particular file in its file system. The file is specified by a URL in the first line of the headers section of the request. The Post method, on the other hand, does allow a data section to be included in the message request. Again, the data must be characterized by the Content-Type and Content-Length fields in the headers section.

Both methods can return the same type of responses. Most responses include a data section—usually some HTML code. However, the data section is optional. The advantage of the Get method is that it is simple. In fact, all hypertext links you see in web pages use the Get method. It is not possible to specify the Post method in a hypertext link. However, the Get method does limit the amount of data that can be transmitted to the server. All data must be included as part of the URL requested. The length of URL's is limited to 256 characters. So for long requests, the Post method is preferred. Table 3.3.1 and Table 3.3.1 show two HTTP connections. One uses the Get method and the other uses the Post method. They both communicate the same information to the server and get the same response. Notice how the different methods communicate the information.

Get Method	
Request	GET /?field1=value1&field2=value2&field3=value3 HTTP/1.0
Response	HTTP/1.0 200 OK MIME-Version: 1.0 Server: CERN/3.0 Date: Wednesday, 19-Aug-98 18:43:14 GMT Content-Type: text/html Content-Length: 73 Last-Modified: Wednesday, 12-Aug-98 20:52:50 GMT <html> <title>Test Page</title> <body>This is a test page.</body> </html>

Table 3.1: Example use of the HTTP Get method

Post Method	
Request	POST / HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 41 field1=value1&field2=value2&field3=value3
Response	HTTP/1.0 200 OK MIME-Version: 1.0 Server: CERN/3.0 Date: Wednesday, 19-Aug-98 18:43:14 GMT Content-Type: text/html Content-Length: 73 Last-Modified: Wednesday, 12-Aug-98 20:52:50 GMT <html> <title>Test Page</title> <body>This is a test page.</body> </html>

Table 3.2: Example use of the HTTP Post method

Our Messaging Protocol uses the HTTP Post method to send its messages. Our messages are contained entirely within the data section of the HTTP message.

All messages, both request and response, are encoded using the application/x-www-form-urlencoded MIME type. This is the type most commonly used with the Post method. It defines the data as a set of fields. Each field is separated by a the '&' character. Field name and field value are separated by the '=' character. And there is a scheme for escaping special characters.

LOGOUT Message	
Logical	COMMAND=LOGOUT SESSION-ID=6004
Encoded	COMMAND=LOGOUT&SESSION-ID=6004

Table 3.3: Example message in logical and actual formats

We define one special required field in all our messages. This is the field named COMMAND. It contains the name of the message being sent. The value of the COM-

MAND field also infers what other fields in the message are required and optional. Fields are not required to be in any particular order. The COMMAND field, for example, might be first, last, or somewhere in the middle. Also, if a field cannot be used or cannot be understood, it is ignored. This way our messages can be at least somewhat backwards compatible.

3.3.2 Reflection

We want to be able to facilitate interaction between clients and servers that might not necessarily support exactly the same set of messages. So our protocol allows a client to discover at runtime which messages the server supports. This process is called reflection. Reflection is accomplished in the Messaging Protocol by using the MESSAGE-LIST-REQUEST and MESSAGE-LIST messages. The former is the message the client sends to the server to request a list of supported messages. The latter is the response the server sends back to the client along with a list of the messages it supports.

Reflection	
Request	COMMAND=MESSAGE-LIST-REQUEST
Response	COMMAND=MESSAGE-LIST MESSAGES=LOGIN, LOGOUT, POLL

Table 3.4: Example use of reflection messages

Once it has received the list of messages that the server will accept, the client should limit the messages it uses to those supplied in the list.

3.3.3 Session Control

For various reasons, the server needs to keep track of the clients that are currently connected to it. However, HTTP uses stateless connections. So we need to build some kind of mechanism on top of HTTP that will allow our server to track clients.

To accomplish this, the server forces each client to log in before it can send any other messages. When a new client logs in, the server assigns it a unique session identification number. All subsequent messages between the client and server must include this session ID in a field called SESSION-ID. Once a client has logged in, the server keeps a reference to its session ID until either the client explicitly logs out or the server restarts.

Session Control	
Request	COMMAND=LOGIN
Response	COMMAND=ACK SESSION-ID=6004 MESSAGE=LOGIN
Request	COMMAND=*request1* SESSION-ID=6004 *request1 fields*
Response	COMMAND=*response1* SESSION-ID=6004 *response1 fields*
Request	COMMAND=*request2* SESSION-ID=6004 *request2 fields*
Response	COMMAND=*response2* SESSION-ID=6004 *response2 fields*
Request	COMMAND=LOGOUT SESSION-ID=6004
Response	COMMAND=ACK SESSION-ID=6004 MESSAGE=LOGOUT

Table 3.5: Example use of session control messages

3.3.4 Polling

Sometimes the server needs to send a message to a client even though the client has not made a direct request for the information. Because we use HTTP as a basis for a our protocol and HTTP requires that the client initiate all connections, we need

to provide our own mechanism to allow the server to send unsolicited messages to the client. To solve this problem, our protocol requires that each client polls for new messages every 5 seconds. If there are messages waiting for the client, the server will send them back in response to the poll.

Polling is also how we implement session timeouts. If the client does not poll every 5 seconds, the server will assume the client is no longer active. It will then automatically log the client out of the system. In actual fact, this 5 second time limit is a conservative lower bound on the time the server allows between polls. If the client misses this mark by a second or so, it will most likely not get logged out. We do this to prevent client implementations from being forced to give polling top priority. A poll can always wait for other message requests to complete.

3.3.5 Data Transfer

Our message format does not really lend itself well to sending large amounts of binary data over a network. For this reason, when sending data such as images and graphs over our protocol, we do not use messages. In fact, we stop using the HTTP Post method all together. Instead, we use the HTTP Get method.

Data Transfer	
Request	GET /images/image1.gif HTTP/1.0
Response	HTTP/1.0 200 OK MIME-Version: 1.0 Server: CERN/3.0 Date: Wednesday, 19-Aug-98 18:43:14 GMT Content-Type: image/gif Content-Length: 1002928 Last-Modified: Wednesday, 12-Aug-98 20:52:50 GMT *binary image file*

Table 3.6: Example data transfer request and response

To get data from the server, a client must first determine the URL of the data it

wants to get. The user gets a URL by using one or more of the Data Control messages (see Section 3.4) However, to get the actual data the client needs to connect to the server using the Get method and request the URL directly.

3.4 Messages

This section briefly describes each major group of messages currently supported by the MEMS Characterization System. We refer to a message by its command name. The command name of a message is also the value of its COMMAND field. Except for a few special cases, message behavior can be represented as request/response pairs. The client sends out a request message and the server returns the appropriate response message. For a complete reference of all the messages supported, see Appendix A.

3.4.1 Generic

Generic messages perform simple protocol functions. They function independently of sessioning rules. So a client does not need to login to a server in order to use a generic message. Currently, there is only one generic message the client can send: MESSAGE-LIST-REQUEST.

Generic Messages	
Request	Normal Response
MESSAGE-LIST-REQUEST	MESSAGE-LIST

Table 3.7: List of possible generic messages and their responses

3.4.2 Session Control

There are three session control messages available in the current system. There is a message to allow a client to log in. There is another message to allow a client to log out. Finally, there is the poll message that the client must send every 5 seconds to

keep its session active. Together, these messages help the server to trace an individual client across multiple connections.

Session Control Messages	
Request	Normal Response
LOGIN	ACK
LOGOUT	ACK
POLL	ACK

Table 3.8: List of possible session control messages and their responses

When a client logs in, it receives a session ID unique to its session. It must then include that session ID in every subsequent message it sends to the server. The session ID gets put in a special field called SESSION-ID. Almost every message requires a SESSION-ID field. Those that do will can an error to be returned if used without a valid session ID. A client can free up its session ID by logging out. Clearly, after logging out the client can no longer use most of the features of the server. To keep its session active and to prevent and unwanted automatic logout, the client must poll the server every 5 seconds.

The polling message also allows the server to send information to a client that the client did not explicitly request. We do this by returning waiting messages in response to the poll from the client.

3.4.3 Hardware Control

Hardware control messages affect the state of the hardware devices attached to the server. These devices include a microscope, a frame grabber, a camera, a strobe pulse generator, and a stimulus waveform generator. The hardware control messages allow clients to manipulate these devices remotely. For example, a client might send a message telling the server to move the microscope stage up 2 millimeters.

However, the server only allows one client at a time to control the hardware. We

Hardware Control Messages	
Request	Normal Response
TAKE-CONTROL	ACK
CEDE-CONTROL	ACK
GET-CONTROL-STATE	SET-CONTROL-STATE
SHUTDOWN	ACK
RESET	ACK
SET-STROBE	ACK
GET-STROBE	SET-STROBE
SET-STAGE	ACK
GET-STAGE	SET-STAGE
SET-FOCUS	ACK
GET=FOCUS	SET-FOCUS
AUTOFOCUS	SET-FOCUS
SET-STIMULUS	ACK
GET-STIMULUS	SET-STIMULUS
SET-MAGNIFICATION	ACK
GET-MAGNIFICATION	SET-MAGNIFICATION
SET-WAFER	ACK
GET-WAFER	SET-WAFER
CALIBRATE	ACK
SET-SERIES-PARAMETERS	ACK
GET-SERIES-PARAMETERS	SET-SERIES-PARAMETERS
CAPTURE	CAPTURE-RESULT

Table 3.9: List of possible hardware control messages and their responses

define a set of messages that allow clients to view and change a piece of state we call the Control Status. For each client, the Control Status may take on of three values: ACTIVE, PASSIVE, or NONE. The value ACTIVE means that particular client has control of the hardware. The value PASSIVE means some other client has control. If no client has control, all clients show their Control Status value as being NONE. There is one message that allows clients to view their Control Status. It returns one of the three values above in its response. There are also two messages that allow a client to take control of the hardware and give it up.

If a client whose Control Status is not currently ACTIVE tries to change the state of some hardware device, the server will generate an error.

3.4.4 Data Control

Data control messages are messages that affect data organization in the system. A client can perform two basic functions with the server's database. It can of course retrieve data. However, it can also modify the database. A client can modify data in several ways. It might actually change some part of existing data. Or it might add new data to the database. Finally, a client can actually control where data is stored in our MEMS Characterization System. Our system allows us to distribute data between multiple servers. Clients have control over which server or servers store what data. They can replicate, copy, or delete data

Data Control Messages	
Request	Normal Response
SELECT-DATA-IDS	DATA-ID-LIST
GET-DATA-ID	META-DATA
CREATE-DATA-ID	NEW-DATA-ID
UPDATE-DATA-ID	ACK
DELETE-DATA-ID	ACK
REPLICATE-DATA-ID	ACK
MOVE-DATA-ID	ACK

Table 3.10: List of possible data control messages and their responses

We define seven data control messages to allow a client to do all these things with the server's data. We have two messages, SELECT-DATA-IDS and GET-DATA-ID, to identify and retrieve data from the server. Three more messages, CREATE-DATA-ID, UPDATED-DATA-ID, and DELETE-DATA-ID, control data creation, modification, and destruction respectively. Finally, we include two messages, REPLICATE-DATA-ID and MOVE-DATA-ID, to control where particular pieces of data are stored within a set of MEMS Characterization Servers.

3.4.5 Processing Control

Processing control messages give instructions to the Data Processing Engine. A Data Processing Engine takes in some data and some parameters telling it how to process

the data. We define one processing control message, called PROCESS, to send these arguments to the Data Processing Engine on the server. The message has one field to specify the data ID of the data to pass into the engine. It also has another field to specify the processing parameters to pass in along with the data.

Processing Control Messages	
Request	Normal Response
PROCESS	PROCESS-RESULT

Table 3.11: List of possible processing control messages and their responses

When the engine is done working, it returns a new piece of data. The server returns the data ID of this new data in a field of the PROCESS-RESULT message.

As the Data Processing Engine is an optional part of the server, processing control messages are likely to not be supported by some servers. A client should always check the server's list of supported messages before performing and server-side data processing.

3.4.6 Inter-Client Communication

A client will sometimes want to communicate with other clients logged in to the server. So we define a set of messages for inter-client communication.

Inter-Client Messaging Messages	
Request	Normal Response
SESSION-LIST-REQUEST	SESSION-LIST
SEND-MESSAGE	ACK

Table 3.12: List of possible inter-client messages and their responses

To facilitate this inter-client communication we introduce the idea of public and private session ID's. The session ID described above is actually a private session ID. It is given only to the client associated with the session. Every private session

ID, however, has a corresponding public session ID. Only the server knows which public ID corresponds to which private ID. So the public ID's can be broadcast to all clients. The server need not fear that a greedy client might masquerade as belonging to another session.

When a client wants to contact another client, it asks the server for a list of current public session ID's. It then chooses which ID it wants to communicate with. It sends a message to the server that includes the communication, its own public session ID, and the receiving client's public session ID. It is the server's responsibility to see that the communication is sent on the appropriate client the next time it polls for new messages.

3.4.7 Errors

An error message may be returned by the server as an alternative response to any request message. There are two specific error messages. The first tells the client that it failed to supply a valid session ID with a request message that required one. The second tells the client that it tried to modify the state of some hardware on the server without the proper Control Status. There is also a generic error message that can be used in any situation. Finally, there is a busy message that tells the client that the server is not accepting messages and to try again later.

Error Messages
ERROR
INVALID-SESSION-ID
CONTROL-ERROR
BUSY

Table 3.13: List of possible server error messages

3.5 Example Session

This section details a simple client session using the Messaging Protocol. It provides a complete view of client server interactions for the entire session, down to the HTTP message text for each connection.

Our sample session has three parts. First, the client logs in and gets a new session ID. Then, it polls the server once with its new ID. Finally, it logs out and ends its session. In our example, the server has no messages waiting for the client when polling occurs.

Example Session	
Client	POST / HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 13 COMMAND=LOGIN
Server	HTTP/1.0 200 OK Content-Type: application/x-www-form-urlencoded Content-Length: 41 COMMAND=ACK&SESSION-ID=6004&MESSAGE=LOGIN
Client	POST / HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 28 COMMAND=POLL&SESSION-ID=6004
Server	HTTP/1.0 200 OK Content-Type: application/x-www-form-urlencoded Content-Length: 40 COMMAND=ACK&SESSION-ID=6004&MESSAGE=POLL
Client	POST / HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 30 COMMAND=LOGOUT&SESSION-ID=6004
Server	HTTP/1.0 200 OK Content-Type: application/x-www-form-urlencoded Content-Length: 42 COMMAND=ACK&SESSION-ID=6004&MESSAGE=LOGOUT

Table 3.14: Example session for the MEMS Characterization System

Chapter 4

Messaging Subsystem

4.1 Overview

Our server implementation is divided into several modules. These modules do not talk to each other using messages. So we need some way of translating information from message form into procedure calls. Our server groups all of this functionality into one module: the Messaging Subsystem. It is the only module in the server that understands the Messaging Protocol described in Chapter 3. It provides all the utilities the server will need to communicate with clients in the MEMS Characterization System.

4.2 Description

The off-the-shelf web server implements basic HTTP services. It also provides the Java Servlet developer interface that we write the rest of our server against (see Section 4.3 below). Furthermore, in the future we may choose to use some security features of the web server to provide encryption and authentication services to our system. The web server we have chosen to develop with is the Java Web Server from Sun[21]. It is free for non-commercial use. It has nice security features such as an implementation of SSL. And, most importantly, it works very well with Java Servlets. Many other web servers we tried could not run even the simplest of Servlets.

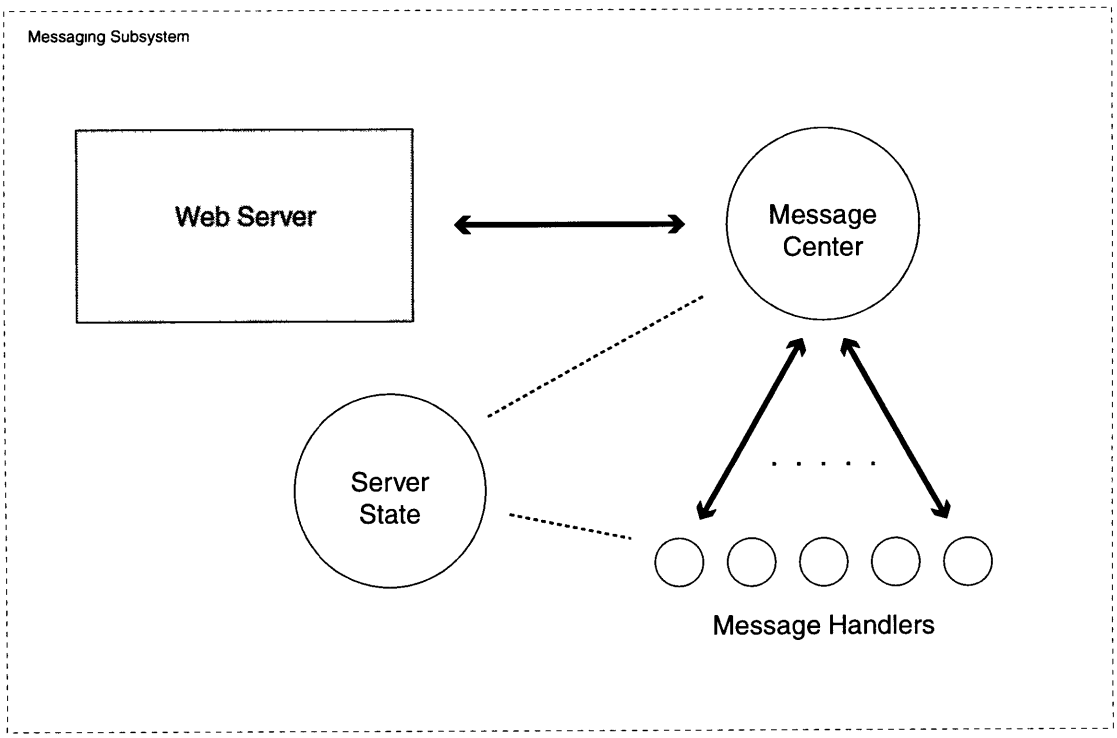

Figure 4-1: Modules comprising the Messaging Subsystem

Currently, when a client connects to the web server it generates a call to the Servlet that implements our Messaging Subsystem. The Messaging Subsystem is composed of several submodules (see Figure 4-1). The submodule that handles all incoming messages is called the Message Center. It parses each request, does some basic error checking, and then passes duties on to the appropriate Message Handler.

Each message, as listed in Chapter 3, has a corresponding Message Handler to process it. Quite simply, a Message Handler is responsible for completing error checking tasks on the message and then generating an appropriate response to the message. It may have to use any number of server utilities and modules to determine the appropriate response.

To help Message Handlers and other modules in the server respond to requests, the Messaging Subsystem contains a submodule called the Server State module. It contains all state information that may be commonly used by the various modules in the system. All server modules have access to the Server State module.

4.3 Java Servlets

The Java Servlet API was developed by JavaSoft and released as part of the Java Development Kit (JDK) starting with JDK version 1.2[20]. A Servlet is a server-side script that runs on the back end of a web server. When the web server gets a request for the URL of a Servlet, it passes the request on to the appropriate Servlet code. The Servlet handles the request in any way it wants to and returns a response through the Servlet API. The web server then passes that response back to the client.

A Servlet is very similar to a CGI script written in Perl or C. The major differences are that these scripts are written in Java and they can be persistent. Java is a simpler language than C and a more general language than Perl. As such, Servlet developers can develop web-based applications that are easier to code than with C CGI. And they can use a greater tool set than is available for Perl CGI.

4.4 Message Handlers

A Message Handler implements a simple interface. Figure 4-2 shows the basic methods exported by a Message Handler. A handler's primary function is to take in a request message and return an appropriate response message. However, a Message Handler also needs to perform some basic error checking on the incoming message. Specifically, it is responsible for determining whether all required fields are present in the request message. It is also responsible for handling all errors generated by other modules it might call.

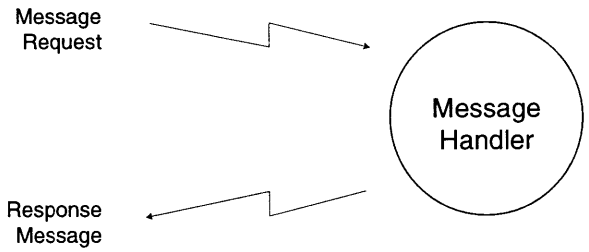

Figure 4-2: Basic Message Handler interface

There is one Message Handler in our server for each request message that the

server supports. To enable the server to support a new message, we must simply write and install a new message handler. No other message handlers are affected by this process.

As an example, we might want to implement a Message Handler for the LOGIN message. When given a LOGIN message, this handler must create and register two new session ID's for the client. It must then return the private session ID to the client in the ACK message. This all assumes that there are no errors in the message. However, there are no required fields in a LOGIN message other than the COMMAND field, so there is not much error checking to be done.

To implement the main function of the handler, we generate two random numbers, one to be a public ID and the other a private ID. Using the Server State module described in Section 4.6, we check that each ID is unique. If one or the other is not, we generate a new random number and test it. We continue until we get two unique ID's. We then register them with the Server Session module. Finally, we build an ACK message with the new private session ID in the SESSION-ID field and the value LOGIN in the MESSAGE field. We end by returning this message. Of course we have to deal with the possibility that we cannot find unique session ID's. In that case, we would return an ERROR message describing the problem.

Our error checking function for the LOGIN message consists simply of a test to make sure that the value of the COMMAND field in the incoming message is LOGIN. If this test passes, we have a valid message.

4.5 Message Center

We group all our Message Handlers together in a submodule we call the Messaging Center. Figure 4-3 demonstrates the relationship between the Message Center and its constituent Message Handlers. Before using the Message Center, our server must register Message Handlers with it. Our server must register a handler for each message it wants the center to be able to handle.

When a message comes in, the Message Center finds the appropriate Message

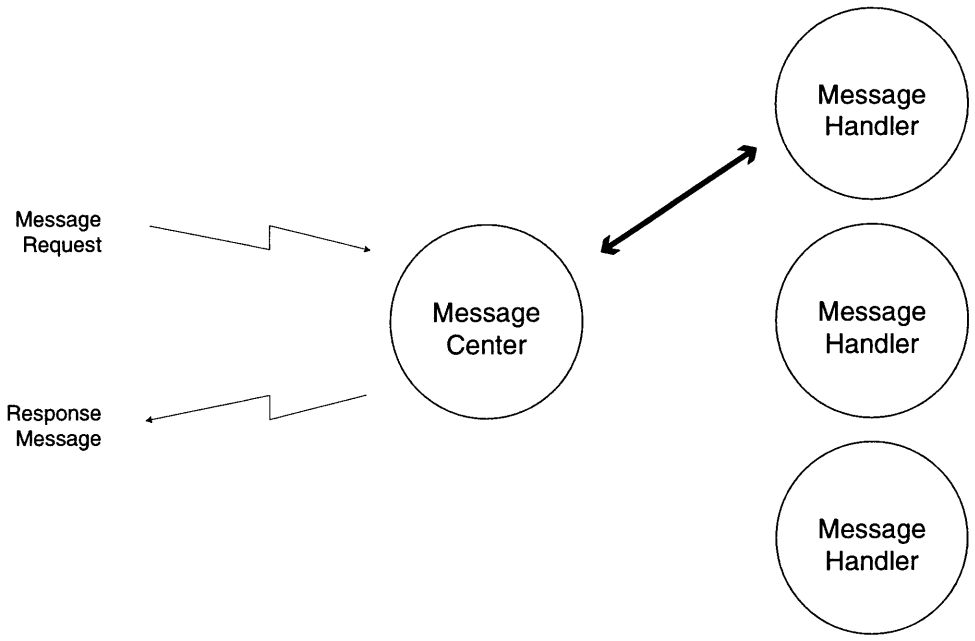

Figure 4-3: Relationship between Message Handlers and the Message Center

Handler. It checks the message's validity using the Message Handler's own error checking function. It also checks to make sure that any private session ID supplied in the SESSION-ID field is valid. It then passes off control to the Message Handler's main function. If an error occurs somewhere in the process, the Message Center returns an error message to the client.

The error checking that the Message Center does is limited to validating private session identification numbers, if they exist. Some messages do not require session ID's. In the case when one does exist, however, the Message Center will check it against the private session ID's of existing sessions.

The only other thing the Message Center does is generate a list of the messages it supports. This list is comma-delimited. It is needed specifically for use by the MESSAGE-LIST-REQUEST Message Handler. This handler must return the list of supported messages in its response message to the client.

4.6 Server State

There are several complex interconnections between the various submodules in the Messaging Subsystem and the server as a whole. They share many pieces of state information. We group these pieces together into one global submodule called Server State. The components of the Server State submodule are: the Session List, the Outgoing Message Queue, and four references to other modules.

The Session List and Outgoing Message Queue are actually integrated parts of the Server State submodule. When the server needs something from either of these two components it asks the Server State module for the information. The Server State module acts as a middle-man. It gets the information from the appropriate component and returns it to the server.

However, to access any other piece of state information, such as the current settings of the microscope, the server does not talk to the Server State submodule directly. Instead, it asks the submodule for a reference to some other module. In the case of microscope settings, it would ask for a reference to the Hardware Control Module. It would then ask the Hardware Control Module directly for the state of the microscope.

4.6.1 Session List

The server needs to maintain a list of all active sessions. For the purposes of inter-client communications, it needs to maintain two session ID's per session—a private ID and a public ID.

The private ID is given only to the client that initiates the session. That client must include its private ID in the SESSION-ID field of all subsequent request messages.

The public ID, however, can be given to any client. It is primarily used to identify which session an inter-client message should be sent to. When it receives an inter-client message, the server can match up public and private ID's and send the message on to the appropriate client.

If the server were to broadcast private ID's, however, a malicious client could

steal the session ID of another client. If victim were in control of the hardware on the server, for example, the attacking client could then steal control away from it. Or it might simply change some piece of hardware state without really having the authorization to do so.

On top of storing all active private and public session ID's, the Session List also keeps track of which client, if any, currently controls the server's hardware. Any other module can ask it for the public or private session ID of the controlling client. Modules can also ask the Session List to create a new session or close an active one. These to capabilities are used to implement the session control messages such as LOGIN and LOGOUT.

4.6.2 Outgoing Message Queue

The Outgoing Message Queue provides the mechanism to send a message to a client when the client did not explicitly ask for a message. The nature of HTTP makes it such that the server cannot initiate a connection. All messages from the server must be in response to requests from the client. To get around this shortcoming, we require that the client poll the server every 5 seconds to see if it has any messages waiting for it. This polling also helps the server keep track of active sessions and close inactive sessions.

Our Outgoing Message Queue is useful in many different scenarios. If an inter-client message, for example, comes in to the server, it will be queued up in the Outgoing Message Queue. Each session has its own independent queue. The next time the right client polls, it will get the inter-client message as a response. Alternatively, the queue can be used to inform all clients of changes in the state of the server's hardware. When the controlling client moves the microscope stage, for example, the server can queue up SET-STAGE messages for all other clients so that they know what has happened.

The Outgoing Message Queue is simply a collection of queues for the individual sessions. Queues are identified by their corresponding private session ID. We use the private session ID as an identifier because that way it can be quickly matched up

with its associated client when the client polls.

4.6.3 Module References

All of the modules have to talk to the other modules in the server at one time or another. To simplify this task, we collect references to each module in the Server State submodule. The modules referenced by the Server State submodule are the Database Subsystem, Image Analysis Engine, Hardware Control Module, and Message Center.

So, for example, when a client requests the current x-axis position of the microscope stage, the GET-STAGE Message Handler can use the Server State submodule to help it out. It will ask for a reference to the Hardware Control Module. It will then ask that module for the current x-axis position and return the correct SET-STAGE message to the client.

It is important to note that state such as current hardware settings and current database configuration is not stored in the Server State submodule. Rather, the Server State submodule contains references to the modules that do contain this information. We do this to avoid duplicating state information and having to deal with the resulting concurrency issues.

Chapter 5

Database Subsystem

5.1 Overview

The MEMS Characterization System deals with a large volume of data. Each of the server components needs to access this data some of the time. So, the server needs some way to store, organize, and distribute data. We call this module the Database Subsystem.

5.2 Description

The Database Subsystem is divided into two parts: a Meta-Data Store and a Data Store. The Data Store holds the actual data of interest, such as an image or a graph. The Meta-Data Store, on the other hand, holds information about the pieces of data in the data store. Each element in the Meta-Data Store refers either to a group of other elements in the store or to one element in the Data Store. Each element in the Data Store, on the other hand, has exactly one element in the Meta-Data Store that refers to it.

5.3 Java Database Connectivity (JDBC)

The Java Database Connectivity (JDBC) API is a part of the Java Development Kit[2]. It allows a Java program to connect to any SQL database. By using JDBC, the code to connect is not written specifically for the database being connected to.

To connect to a database, a program must load a JDBC driver written specifically for the brand of database it wishes to connect to. For example, a program would load a different JDBC driver to connect to an Oracle8 database than it would to connect to an mSQL database[11]. However, each driver implements the same Java interface. So the code used to connect to one database will work with any other database.

5.4 Meta-Data Store

We call the information the Meta-Data Store keeps data elements. All data elements have a unique data identification number, a text description, and a creation date. Each data element, however, is also one of two other data types. The first type is called meta-data. The second is called a data set.

All meta-data elements contain information about data in the Data Store. This information is the format the data is stored in and the URL of the data.

Data sets, on the other hand, define the hierarchy of the data in the entire Database Subsystem. This data hierarchy tells us how the individual pieces of data relate to one another as sets, and sets of sets, etc. For example, we might capture a run of five related images using the microscope and camera on the server. We can store these images as one set in the database. The images themselves would be stored in the Data Store. The data format and a URL referencing the data would be stored as meta-data in the Meta-Data Store. But the information relating the images to one another would be stored as a data set in the Meta-Data Store.

We choose to implement our Meta-Data Store using a relational SQL database. A relational database allows us to define our data hierarchy in a straightforward manner. The relational database gets its name from the way it stores information.

Information in a relational database is organized in many different tables. Tables, and the information they contain, are related to one another by the fields they share in common.

In our implementation, the Meta-Data Store has three tables. A main table describes all data elements. This includes a field indicating whether each data element is also a data set or a piece of meta-data. The two additional tables provide the extra information associated with each type. One table describes the data sets, while the other describes the meta-data.

The main table contains four fields. The first is the data ID for each element. The second is the date each was created. The third is a text description of the data element. And the fourth field indicates whether each element is meta-data or a data set. This fourth field proves to be redundant, as the information it contains can be inferred by the relationship of the three tables. However, it can simplify some data queries.

The meta-data table contains three fields. The first field is the data ID of the element. The second is the data format. And the third field is the URL that refers to a piece of data in the Data Store. This URL can be used by the client to request the data directly from the web server.

The data set table contains just two fields. Both fields contain data ID's. The data element referred to by the first field is a data set containing the data element referred to by the second field. The same data ID may appear multiple times in either column of the table. To specify that a data set contains three data elements, for example, we would put three entries in the data set table. Each entry would have the same value in the first field. But each of the second fields' values would be a different member of the data set.

To access our relational database, we use a JDBC connection. For the most part, the Database Subsystem does not export interfaces using SQL as arguments. The only exception is the method to support the SELECT-DATA-IDS message. This message has a SQL SELECT statement as its argument. So the method supporting it passes this SQL statement into the database. The statement is constructed by the

client. Therefore, the client must understand the basic structure of the relational database. The response generated by the query, however, is parsed by the Database Subsystem and not return as a raw SQL result set. Rather, the module returns a list of data ID's that matched the query.

The other Meta-Data Store functions supported by the Database Subsystem are meta-data lookup, creation, modification, and deletion. Lookup returns all the information about the particular data element requested. This does not include the information for any elements above it or below it in the data hierarchy. Creation takes in a series of arguments that correspond to all the data fields in the relational database. It returns the data ID of the new data element. Modification is similar to creation, except that all its arguments are optional and it does not return anything. Which arguments are included defines which data fields get modified. Finally, deletion takes in a data ID and removes the corresponding data element from the Meta-Data Store. It does not, however, remove any data elements above it or below it in the data hierarchy.

Finally, we have a database server sitting behind our JDBC API. The database server itself is an off-the-shelf product. For development purposes, we are using a mSQL server. But the MEMS Characterization System would work equally well with an Oracle8 database or a DB2 database.

5.5 Data Store

Compared to the Meta-Data Store, our Data Store implementation is very simple. It is just a flat directory of files. All hierarchy is defined by the Meta-Data Store, so none is needed in the Data Store.

The data URL in every element of meta-data refers to a single element in the Data Store. To retrieve something from the Data Store, a client simply needs to ask for a URL it finds in the Meta-Data Store.

To do this, however, the client does not send a message to the server. Messages use the HTTP Post method. Retrieving data from the Data Store requires the client

to use the HTTP Get method to request the URL.

5.6 Distributing the Database

There are two possible ways to distribute the database in the MEMS Characterization System. The first distributes only the meta-data. The second distributes the entire database.

5.6.1 Distributing Meta-Data

The administrator of the MEMS Characterization System may choose to distribute just the meta-data in his implementation. Such a design would allow servers to retrieve remote meta-data through their local Database Subsystem. However, the actual Data would not be replicated. So a client would have to make a direct request to the remote server in order to retrieve Data. Making such a request also requires that the client log in to the remote server.

To implement such a system, all the administrator need do is reconfigure the SQL database server he is using for the Meta-Data Store. He will have to distribute the database and administer it himself. Many commercial database servers support data replication.

5.6.2 Distributing Data and Meta-Data

The other option available to the administrator is to distribute the entire database. This is a build option in the MEMS Characterization Sever that basically turns the server into a client as well.

The mechanism is already in place for any client to copy both meta-data and data from the server. So by making the server act as a client, replicating information is simple.

To allow control over where particular data and meta-data is physically stored, we define two new messages specifically for use in distributing data. One lets us

replicate, or copy, data. The other lets us move data. Both messages operate on both data and meta-data at the same time. If the data ID supplied refers do a data set, just the one data element is replicated. However, if the data ID refers to meta-data, both the meta-data and the data it is associated with is replicated.

Unfortunately, because of the way our system is designed, all replication and move requests must be sent to the server data is to be replicated to. This stems from the fact that the receiving server is the one that will have to act as the client. Again, in our system, the client must always initiate a connection.

Chapter 6

Hardware Control Module

6.1 Overview

At the heart of our entire system is a set of hardware components that work together to take pictures of MEMS devices. In order to control this hardware with a computer we need to write some native code. Whatever type of machine the server is running on, we need to be able to access device drivers to control our hardware. Unfortunately, we cannot do this directly from Java. Rather, we must use native code. Our MEMS Characterization Server groups all of this machine-specific code into a module called the Hardware Control Module.

6.2 Description

The Hardware Control Module has two related functions. First, it exports methods to change the state of the various hardware devices attached to the server. Second, it allows other modules to query for the current state of each device.

The Hardware Control Module does not, however, worry about which client has control of the hardware at any particular time. It lets the Messaging Subsystem take care of that. The Hardware Control Module never refuses to change the state of a hardware device when asked.

The Java part of the Hardware Control Module is just a thin wrapper around

native interfaces. However, because of this wrapper, other modules in the system treat the Hardware Control Module just as if it were written entirely in Java.

6.3 Java Native Interface (JNI)

The Java Native Interface (JNI) is an API developed by JavaSoft and included in the JDK since JDK version 1.1[19]. It allows a Java program to call code native to the machine it is running on.

To do this, the developer defines in Java the methods to be implemented using native code. He specifies that a method is native by adding the native keyword to the prototype. The developer does not provide a body for the native methods.

After compiling his Java code, the developer then uses a Java utility to generate a C and C++ header file. The file defines the appropriate prototypes for the native methods. By including the new header and another JNI-specific C header, he can implement the methods.

Finally, the developer compiles his native code to a shared library. At run time, the Java Runtime Environment (JRE) will load the library when a native method is requested.

6.4 Java Interfaces

The Java interfaces in the Hardware Control Module are the methods that other Java modules call. They form the outside of the wrapper of code that makes up the module.

The interfaces simply look like programmatic translations of the hardware control messages in our Messaging Protocol. For example, the module exports a method that mimics the SET-FOCUS message. The message takes in a single number as its parameter. The response to the message returns no new information to the client. So, the corresponding method in the Hardware Control Module takes in a single number and returns nothing.

6.5 Native Interfaces

In contrast to their Java counterparts, the native interfaces form the inner wall of the thin wrapper that is the Hardware Control Module.

These methods look very similar to the Java interfaces. However, we do some data translation to convert between objects and primitive data types when possible.

Chapter 7

Data Processing Engine

7.1 Overview

The most important thing our MEMS Characterization System does is capture images of MEMS devices. But after doing this, a user will want to analyze the data they have collected. So the second most important part of the system is the Data Processing Engine. Its job is to automate the process of analyzing the large amounts of data users collect about any one particular MEMS device.

7.2 Description

As with the Hardware Control Module, the Java code for the Data Processing Engine simply forms a thin wrapper around the more substantial native code. All of the functionality of the engine is embedded in the native code itself.

The code that fills our wrapper would be very similar to the code implementing the analysis tools used with ECK. In fact, our engine might be a library of analysis routines. The user could specify which routine to use in part of the parameter string.

This module is an optional component of a MEMS Characterization System. Users may not care to process data on the server. Instead, they might download data to a client machine and process it there. So, we make this module easy to disable and remove.

7.3 Java Interfaces

There is really only one method that a Data Processing Engine exports. This method tells the module to process a set of data using a certain parameter string. The method returns a new data set.

A data set can be a single image, a set of images, a set of sets, and so on. In fact, our data need not be just images. We can have graphs in our system, both as individual data elements and as parts of data sets.

7.4 Native Interfaces

The native interfaces in the Data Processing Engine form the inner side of a Java wrapper. However, they are fundamentally no different from the Java interfaces on the outside. So, the code to translate from one to the other is very simple.

Chapter 8

Conclusion

Current tools available to analyze MEMS devices are incomplete and very clumsy. Engineers using these systems inevitably end up with volumes of data that they do not know what to do with. One reason for this problem is that current tools are very poorly integrated. An engineer must use one tool for image capture and one tool for image analysis. Also, the user interfaces make managing tools and information very difficult.

The work presented in this thesis implements the server side of a remote analysis system for MEMS devices. This system addresses the problems with current analysis tools with an integrated architecture and a more friendly user interface. Image capture and image analysis tools are part of the same software package. Also, they can be accessed from the same client.

The MEMS Characterization System uses a well-defined messaging protocol to facilitate client/server communication. The abstraction barrier this protocol represents allows for the independent development of various different client and server implementations.

The system's server architecture allows for many different configurations. For example, one server implementation might not include any tools to analyze images. Another server might include analysis tools but leave out image capture routines. These two servers might work together over a network to share the workload of MEMS analysis.

The primary goal of this research was to build a first-generation server for MEMS Characterization that could be used as a reference to implement a client. A basic server has been implemented. It adheres to the system's Messaging Protocol and simulates all base functionality. The server will prove invaluable in testing and evaluating future clients. Also, by plugging in complete modules where stubs now exist, future efforts will implement a fully-functional server.

There is, however, one problem with the current MEMS Characterization System architecture. This became apparent through the process of implementing the server. The problem is that HTTP, upon which our Messaging Protocol is layered, requires the client to initiate all communications. While this works well for most situations in our system, there are some cases in which it becomes a real issue. We try to get around this with a polling mechanism. But polling is far less efficient and scalable than a truly bidirectional protocol. Future work may try to develop a modified protocol that is a hybrid between HTTP and some other basic protocol that is actually bidirectional.

Appendix A

Messaging Reference

Messages are listed in request-response pairs. The first message is the client's request, and the second is the appropriate response. Keep in mind that it is always possible that the server is busy and responds with a BUSY message.

A.1 Generic

A.1.1 MESSAGE-LIST-REQUEST

A client can request a list of all the messages the server supports by sending the MESSAGE-LIST-REQUEST message. This message does not require that the client include the SESSION-ID field. However, it is recommended that the client does include the field if possible.

The server returns a comma-delimited list of the messages it supports. The list is put in the MESSAGES field of the MESSAGE-LIST message. The server's response will include a SESSION-ID field only if the client's request included one as well.

Client	
Request	COMMAND=MESSAGE-LIST-REQUEST [SESSION-ID=*sessionID*]
Server	
Normal Response	COMMAND=MESSAGE-LIST [SESSION-ID=*sessionID*] MESSAGES=*messageList*

Table A.1: Logical structure of the MESSAGE-LIST-REQUEST message

A.2 Session Control

A.2.1 LOGIN

The LOGIN message initiates a session between a client and the server. When it receives a LOGIN message, the server registers a two unique session ID's for the new session. One ID is the public session ID, the other is the private session ID (see Section 3.4.6). The server returns an ACK message to the client containing the new private ID in the SESSION-ID field.

Note that the only ID returned to the client is the session's private session ID. The client never needs its own public session ID and is therefore never given it.

Client Request	 COMMAND=LOGIN
Server Normal Response	 COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=LOGIN

Table A.2: Logical structure of the LOGIN message

A.2.2 LOGOUT

The LOGOUT message ends a session between a client and the server. The SESSION-ID field in the request message contains the private session ID of the session to be closed.

When the server receives a LOGOUT message, it frees both the private and public session ID's for the appropriate session. It then returns an ACK message to signal that it is done.

Client Request	 COMMAND=LOGOUT SESSION-ID=*sessionID*
Server Normal Response	 COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=LOGOUT

Table A.3: Logical structure of the LOGOUT message

A.2.3 POLL

The POLL message must be sent by the client every 5 seconds. It serves two purposes. First, it tells the server that the session is still active. Second, it allows the server to send the client any messages it has queued up for it in the Outgoing Message Queue.

In the table below, the first response message represents the case when the server has no messages waiting for the client. However, should there be messages queued up for the client, the server will return the second response message. This response consists of the body of the first message queued for the client with one additional field added on. This field tells the client whether or not there are any more messages queued up for it. If this field is TRUE, the client should send another POLL message immediately to receive the next message. If this field is FALSE, the client need not send another POLL message for 5 more seconds.

The server may only queue up certain messages to send to the client. The following is a list of all the messages a server may queue up:

SEND-MESSAGE
SET-CONTROL-STATE
SHUTDOWN
RESET
SET-STROBE
SET-STAGE
SET-FOCUS
SET-STIMULUS
SET-MAGNIFICATION
SET-WAFER
SET-SERIES-PARAMETERS
CAPTURE-RESULT
PROCESS-RESULT

Client	
Request	COMMAND=POLL SESSION-ID=*sessionID*
Server	
No Messages	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=POLL
Messages	*message body* MORE-MESSAGES=(TRUE | FALSE)

Table A.4: Logical structure of the POLL message

A.3 Hardware Control

A.3.1 TAKE-CONTROL

A client can request control of the server's hardware with the TAKE-CONTROL message. If no client currently has control, the server gives the client control and returns the ACK message. However, if some other client currently has control, the server returns the CONTROL-DENIED message.

If the client successfully gets control of the hardware, it may then proceed to issue state control commands, such as SET-STAGE and SET-STROBE, to the server.

Client Request	COMMAND=TAKE-CONTROL SESSION-ID=*sessionID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=TAKE-CONTROL
Negative Response	COMMAND=CONTROL-DENIED SESSION-ID=*sessionID*

Table A.5: Logical structure of the TAKE-CONTROL message

A.3.2 CEDE-CONTROL

A client can relinquish control of the server's hardware with the CEDE-CONTROL message. When the server has taken control back from the client, it will return the ACK message.

After sending this message, the client may no longer issue state control messages to the server.

Client Request	COMMAND=CEDE-CONTROL SESSION-ID=*sessionID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=CEDE-CONTROL

Table A.6: Logical structure of the CEDE-CONTROL message

A.3.3 GET-CONTROL-STATE

This message is sent by a client to request the current state of its own control over the server's hardware. The server responds with a SET-CONTROL-STATE message. This message includes the CONTROL-STATE field. The CONTROL-STATE field can have the value of ACTIVE, PASSIVE, or NONE. ACTIVE means that the client is in control of the hardware. PASSIVE means that another client is in control of the hardware. NONE means that there is no client in control of the hardware. This last case means that any client is free to take control of the server's hardware.

The SET-CONTROL-STATE message will also be queued up to all clients whenever the hardware control state changes. For example, when a client takes control of the hardware, the server will queue up a SET-CONTROL-STATE message for each of the other clients. Those messages will all have the CONTROL-STATE field set to PASSIVE.

Client	
Request	COMMAND=GET-CONTROL-STATE SESSION-ID=*sessionID*
Server	
Normal Response	COMMAND=SET-CONTROL-STATE SESSION-ID=*sessionID* CONTROL-STATE=(ACTIVE | PASSIVE | NONE)

Table A.7: Logical structure of the GET-CONTROL-STATE message

A.3.4 SHUTDOWN

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client to the server, the SHUTDOWN message tells the server to shut itself down. The server will acknowledge the shutdown command with the ACK message. It will then inform all other clients it is shutting down by queuing up a SHUTDOWN message for each. It will stop responding to messages, instead sending the BUSY message in response to everything. Finally, after it has waited at least 10 seconds to make sure clients can poll for the SHUTDOWN message, it will shut itself down.

When sent by the server, this message tells the client that it is about to shutdown. The client should treat this as being logged off and send no more messages to the server.

Client Request	COMMAND=SHUTDOWN SESSION-ID=*sessionID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SHUTDOWN

Table A.8: Logical structure of the SHUTDOWN message

A.3.5 RESET

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client to the server, the RESET message tells the server to reset all of its hardware state variables, such as stage and strobe settings. After the server has complied, it returns an ACK message as acknowledgment. It then queues up RESET messages for all the other clients logged in.

When sent by the server, the RESET message means that the client in control has reset all the hardware state variables.

Client Request	COMMAND=RESET SESSION-ID=*sessionID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=RESET

Table A.9: Logical structure of the RESET message

A.3.6 SET-STROBE

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-STROBE message changes hardware state variables for the strobe pulse generator. The FREQUENCY argument is always required. Its value is a number in Hertz. The DIVISIONS field is optional. Its value is the

number to divide the stimulus' cycle by to get the strobe's pulse length. Its default value is 8. The PHASE field is also optional. Its value is the division number within the stimulus' cycle to pulse on. Its default value is 0. The number 0 indicates the first division.

When queued up by the server, the SET-STROBE message tells the client which strobe state variables have changed. Only the fields that have changed are included in the message.

Client	
Request	COMMAND=SET-STROBE SESSION-ID=*sessionID* FREQUENCY=*frequency* [DIVISIONS=*divisions*] [PHASE=*phase*]
Server	
Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-STROBE

Table A.10: Logical structure of the SET-STROBE message

A.3.7 GET-STROBE

The GET-STROBE message requests the current value of strobe state within the server. The server responds with a SET-STROBE message. All fields must be included in the response.

Client	
Request	COMMAND=GET-STROBE SESSION-ID=*sessionID*
Server	
Normal Response	COMMAND=SET-STROBE SESSION-ID=*sessionID* FREQUENCY=*frequency* DIVISIONS=*divisions* PHASE=*phase*

Table A.11: Logical structure of the GET-STROBE message

A.3.8 SET-STAGE

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-STAGE message tells the server to change the state of the stage hardware within the server. Only the fields that are included will affect state. All field values are numbers. Rotation numbers are in degrees.

When queued up by the server, the SET-STAGE message tells the client which stage state variables have changed. Only the fields that have changed are included in the message.

Client	
Request	COMMAND=SET-STAGE SESSION-ID=*sessionID* [X-TRANSLATION=*xTranslation*] [Y-TRANSLATION=*yTranslation*] [Z-TRANSLATION=*zTranslation*] [X-ROTATION=*xRotation*] [Y-ROTATION=*yRotation*] [Z-ROTATION=*zRotation*]
Server	
Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-STAGE

Table A.12: Logical structure of the SET-STAGE message

A.3.9 GET-STAGE

The GET-STAGE message requests the current value of stage state within the server. The server responds with a SET-STAGE message. All fields must be included in the response.

A.3.10 SET-FOCUS

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-FOCUS message tells the server to change the state of the focus hardware within the server. The FOCUS-VALUE field contains an integer. The scale for this value is arbitrary and depends largely on the specifics of the hardware.

Client Request	COMMAND=GET-STAGE SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-STAGE SESSION-ID=*sessionID* X-TRANSLATION=*xTranslation* Y-TRANSLATION=*yTranslation* Z-TRANSLATION=*zTranslation* X-ROTATION=*xRotation* Y-ROTATION=*yRotation* Z-ROTATION=*zRotation*

Table A.13: Logical structure of the GET-STAGE message

When queued up by the server, the SET-FOCUS message tells the client that focus state variables have changed.

Client Request	COMMAND=SET-FOCUS SESSION-ID=*sessionID* FOCUS-VALUE=*focusValue*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-FOCUS

Table A.14: Logical structure of the SET-FOCUS message

A.3.11 GET-FOCUS

The GET-FOCUS message requests the current value of focus state within the server. The server responds with a SET-FOCUS message. All fields must be included in the response.

A.3.12 AUTOFOCUS

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

Client Request	COMMAND=GET-FOCUS SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-FOCUS SESSION-ID=*sessionID* FOCUS-VALUE=*focusValue*

Table A.15: Logical structure of the GET-FOCUS message

The AUTOFOCUS message tells the server to run an autofocus routine and adjust the focus value while keeping the other state variables constant. Whatever new value the server comes up with is returned in a SET-FOCUS message.

Client Request	COMMAND=AUTOFOCUS SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-FOCUS SESSION-ID=*sessionID* FOCUS-VALUE=*focusValue*

Table A.16: Logical structure of the AUTOFOCUS message

A.3.13 SET-STIMULUS

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-STIMULUS message tells the server to change the state of the stimulus hardware within the server. Only the fields that are included will affect state. If the value of the SOURCE field is INTERNAL, then any of the optional fields may be included. Otherwise, none of the optional fields are relevant and must not be included. Frequency values are in Hertz, while amplitudes and DC offsets are in millivolts.

When queued up by the server, the SET-STIMULUS message tells the client which stimulus state variables have changed. Only the fields that have changed are included in the message.

Client Request	COMMAND=SET-STIMULUS SESSION-ID=*sessionID* SOURCE=(INTERNAL | EXTERNAL | NONE) [WAVEFORM=(SQUARE | SINE)] [FREQUENCY=*frequency*] [AMPLITUDE=*amplitude*] [DC-OFFSET=*dcOffset*]
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-STIMULUS

Table A.17: Logical structure of the SET-STIMULUS message

A.3.14 GET-STIMULUS

The GET-STIMULUS message requests the current value of stimulus state within the server. The server responds with a SET-STIMULUS message. If the value of the SOURCE field is INTERNAL, all optional fields are included. Otherwise, none of the optional fields are included.

Client Request	COMMAND=GET-STIMULUS SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-STIMULUS SESSION-ID=*sessionID* SOURCE=(INTERNAL | EXTERNAL | NONE) [WAVEFORM=(SQUARE | SINE)] [FREQUENCY=*frequency*] [AMPLITUDE=*amplitude*] [DC-OFFSET=*dcOffset*]

Table A.18: Logical structure of the GET-STIMULUS message

A.3.15 SET-MAGNIFICATION

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-MAGNIFICATION message tells the server to change the state of the magnification hardware within the server. The value in the OBJECTIVE-NUMBER field is a number base 0.

When queued up by the server, the SET-MAGNIFICATION message tells the client that magnification state variables have changed. The new objective number is included in the OBJECTIVE-NUMBER field.

Client Request	COMMAND=SET-MAGNIFICATION SESSION-ID=*sessionID* OBJECTIVE-NUMBER=*objectiveNumber*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-MAGNIFICATION

Table A.19: Logical structure of the SET-MAGNIFICATION message

A.3.16 GET-MAGNIFICATION

The GET-MAGNIFICATION message requests the current value of magnification state within the server. The server responds with a SET-MAGNIFICATION message. All fields must be included in the response.

Client Request	COMMAND=GET-MAGNIFICATION SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-MAGNIFICATION SESSION-ID=*sessionID* OBJECTIVE-NUMBER=*objectiveNumber*

Table A.20: Logical structure of the GET-MAGNIFICATION message

A.3.17 SET-WAFER

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-WAFER message tells the server to change the wafer currently under test. The WAFER-NAME field contains a string that represents the name of the new wafer to be used.

When queued up by the server, the SET-WAFER message tells the client that the wafer has been changed.

Client Request	COMMAND=SET-WAFER SESSION-ID=*sessionID* WAFER-NAME=*waferName*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-WAFER

Table A.21: Logical structure of the SET-WAFER message

A.3.18 GET-WAFER

The GET-WAFER message requests the name of the wafer currently under test by the server. The server responds with a SET-WAFER message. All fields must be included in the response.

Client Request	COMMAND=GET-WAFER SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-WAFER SESSION-ID=*sessionID* WAFER-NAME=*waferName*

Table A.22: Logical structure of the GET-WAFER message

A.3.19 CALIBRATE

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

The CALIBRATE message tells the server to calibrate all of its hardware, if it can be calibrated. When it has finished calibration, the server returns the ACK message.

Client Request	COMMAND=CALIBRATE SESSION-ID=*sessionID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=CALIBRATE

Table A.23: Logical structure of the CALIBRATE message

A.3.20 SET-SERIES-PARAMETERS

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

When sent by a client, the SET-SERIES-PARAMETERS message tells the server to change the state of the image capture routines within the server. Only the fields that are included will affect state. If IS-SINGLE-IMAGE is TRUE, none of the optional fields may be included. Otherwise, any of them may be included. All optional field values are numbers. INITIAL-PHASE tells the server which stimulus division to start capturing images at. This number is based at 0. DELTA-PHASE tells the server how many divisions to increment the phase value with each picture. NUMBER-OF-IMAGES tells the server the number of images to take in the series.

When queued up by the server, the SET-SERIES-PARAMETERS message tells the client which image capture state variables have changed. Only the fields that have changed are included in the message.

Client Request	COMMAND=SET-SERIES-PARAMETERS SESSION-ID=*sessionID* IS-SINGLE-IMAGE=(TRUE | FALSE) [INITIAL-PHASE=*initialPhase*] [DELTA-PHASE=*deltaPhase*] [NUMBER-OF-IMAGES=*numberOfImages*]
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SET-SERIES-PARAMETERS

Table A.24: Logical structure of the SET-SERIES-PARAMETERS message

A.3.21 GET-SERIES-PARAMETERS

The GET-SERIES-PARAMETERS message requests the current value of the series parameters in the image capture routines. The server responds with a SET-SERIES-PARAMETERS message. If the IS-SINGLE-IMAGE field is TRUE, none of the optional fields are included in the response. If it is FALSE, all optional fields are included.

Client Request	COMMAND=GET-SERIES-PARAMETERS SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SET-SERIES-PARAMETERS SESSION-ID=*sessionID* IS-SINGLE-IMAGE=(TRUE | FALSE) [INITIAL-PHASE=*initialPhase*] [DELTA-PHASE=*detlaPhase*] [NUMBER-OF-IMAGES=*numberOfImages*]

Table A.25: Logical structure of the GET-SERIES-PARAMETERS message

A.3.22 CAPTURE

This message may be queued up by the server and sent to a client in response to a POLL message.

Any client sending this message must have control of the server. If it does not have control, the server will return the CONTROL-ERROR message instead of the normal response.

The CAPTURE message tells the server to take an image based on the current hardware settings and series parameters. The server creates a new data ID to represent the image or images. It stores the new data in the database. Finally, it returns a CAPTURE-RESULT message to all clients with the new data ID in the DATA-ID field.

Client Request	COMMAND=CAPTURE SESSION-ID=*sessionID*
Server Normal Response	COMMAND=CAPTURE-RESULT SESSION-ID=*sessionID* DATA-ID=*dataID*

Table A.26: Logical structure of the CAPTURE message

A.4 Data Control

A.4.1 SELECT-DATA-IDS

This message allows a client to get a list of all data ID's in the database that match certain parameters. To specify these parameters, the client includes a SQL SELECT statement in the SQL field of its request message. In order to build the SQL statement, the client must understand the design of the SQL database (see Chapter 5). Also, because the statement must be a SELECT statement, the "SELECT" keyword is assumed and not included.

The server responds to the SELECT-DATA-IDS message with a DATA-ID-LIST message. This message includes in its DATA-IDS field a comma-delimited list of all the data ID's in the database that matched the query.

Client	
Request	COMMAND=SELECT-DATA-IDS SESSION-ID=*sessionID* SQL=*SQLStatement*
Server	
Normal Response	COMMAND=DATA-ID-LIST SESSION-ID=*sessionID* DATA-IDS=*dataIDList*

Table A.27: Logical structure of the SELECT-DATA-IDS message

A.4.2 GET-DATA-ID

The GET-DATA-ID message asks the server to retrieve all information from the database about a particular data element. The data element of interest is specified in the DATA-ID field.

There are two possible responses to the GET-DATA-ID message. The first is returned in the case when the data element is a data set. The second is returned when it is a piece of meta-data. Both messages share the DATA-ID, TIMESTAMP, and DESCRIPTION fields. The TIMESTAMP field includes an integer representation of the time and data when the data element was created. The DESCRIPTION field includes a string description of the data element.

The DATA-SET message also includes a SET-LIST field. This field contains a comma-delimited list of all the data ID's the set includes. In data sets, ordering of data ID's is important. Ordering may specify how a Data Processing Engine should interpret the set.

The META-DATA message includes two additional fields. The first is the URL field. This field's value is the URL of the raw data this meta-data refers to. The second field the META-DATA message includes is the MIME-TYPE field. It's value is the MIME type of the data located at URL.

Client	
Request	COMMAND=GET-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID*
Server	
Data Set	COMMAND=DATA-SET SESSION-ID=*sessionID* DATA-ID=*dataID* TIMESTAMP=*timestamp* DESCRIPTION=*description* SET-LIST=*setList*
Meta-Data	COMMAND=META-DATA SESSION-ID=*sessionID* DATA-ID=*dataID* TIMESTAMP=*timestamp* DESCRIPTION=*description* URL=*url* MIME-TYPE=*mimeType*

Table A.28: Logical structure of the GET-DATA-ID message

A.4.3 UPDATE-DATA-ID

This message is used to change the value of any field or fields associated with a particular data ID. These changes are made in the database on the server.

The client must know whether the data of interest is a data set or a piece of meta-data. It specifies the data of interest with the DATA-ID field. Any optional fields omitted from the message will retain their original values. Also, the SET-LIST field may only be included for data sets. The URL and MIME-TYPE fields may only be included for pieces of meta-data.

When the server has finished updating the database, it returns an ACK message to the client.

A.4.4 CREATE-DATA-ID

The CREATE-DATA-ID message asks the server to create a new data element. The request message includes all the information necessary to create a new element. The IS-SET field contains TRUE or FALSE depending on whether the element is a data set or piece of meta-data. If and only if it is TRUE, the SET-LIST field must be included. If and only if it is FALSE, the URL and MIME-TYPE fields must be included. The client does not provide timestamp or data ID information as it will be created by the server.

When it receives a CREATE-DATA-ID message, the server generates a new data ID and timestamp for the element. It then adds the element to the database. Finally,

Client Request	COMMAND=UPDATE-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID* [TIMESTAMP=*timestamp*] [DESCRIPTION=*description*] [SET-LIST=*setList*] [URL=*url*] [MIME-TYPE=*mimeType*]
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=UPDATE-DATA-ID

Table A.29: Logical structure of the UPDATE-DATA-ID message

it returns a NEW-DATA-ID message to the client. This message includes the new data ID created in its DATA-ID field.

Client Request	COMMAND=CREATE-DATA-ID SESSION-ID=*sessionID* IS-SET=(TRUE | FALSE) DESCRIPTION=*description* [SET-LIST=*setList*] [URL=*url*] [MIME-TYPE=*mimeType*]
Server Normal Response	COMMAND=NEW-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID*

Table A.30: Logical structure of the CREATE-DATA-ID message

A.4.5 DELETE-DATA-ID

This message removes a data element from the database. Which element is to be removed is specified in the DATA-ID field. The server removes only the element specified. If the element is a data set, it does not remove any members of the set. When it is done, the server returns an ACK message.

Client Request	COMMAND=DELETE-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=DELETE-DATA-ID

Table A.31: Logical structure of the DELETE-DATA-ID message

A.4.6 MOVE-DATA-ID

This message can only be used when both the Meta-Data Store and the Data Store are distributed.

This message moves data from one server to another in a distributed database. The data to be moved is specified by the DATA-ID field. The server from which to move the data is specified by the SOURCE field. A source string is the complete URL of another server (e.g. http://sabbath.mit.edu/mems/). Note that this message must be sent to the server to which the data will be moved.

When it receives a MOVE-DATA-ID message, the server connects to the source server using the Messaging Protocol. It copies the relevant meta-data and data to its own database. It then deletes the data from the source server. Finally, it returns an ACK message to the client. All of this is accomplished by the server masquerading as a client itself.

Client Request	COMMAND=MOVE-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID* SOURCE=*sourceServer*
Server Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=MOVE-DATA-ID

Table A.32: Logical structure of the MOVE-DATA-ID message

A.4.7 REPLICATE-DATA-ID

This message can only be used when both the Meta-Data Store and the Data Store are distributed.

This message copies data from one server to another in a distributed database. The data to be copied is specified by the DATA-ID field. The server from which to

copy the data is specified by the SOURCE field. A source string is the complete URL of another server (e.g. http://sabbath.mit.edu/mems/). Note that this message must be sent to the server to which the data will be copied.

When it receives a REPLICATE-DATA-ID message, the server connects to the source server using the Messaging Protocol. It copies the relevant meta-data and data to its own database. Finally, it returns an ACK message to the client. All of this is accomplished by the server masquerading as a client itself.

Client	
Request	COMMAND=REPLICATE-DATA-ID SESSION-ID=*sessionID* DATA-ID=*dataID* SOURCE=*sourceServer*
Server	
Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=REPLICATE-DATA-ID

Table A.33: Logical structure of the REPLICATE-DATA-ID message

A.5 Processing Control

A.5.1 PROCESS

The PROCESS message tells the server to send some data to the Data Processing Engine. The message indicates which data to send with the DATA-ID field. The server sends the data indicated by the data ID in that field to the engine with a string argument. The string argument comes from the PARAMETERS field of the PROCESS message. The engine generates a new data, which the server stores in the database.

The server responds with a PROCESS-RESULT message. This message includes a DATA-ID field to indicate the data ID of the new data generated by the Data Processing Engine.

Also, the same PROCESS-RESULT message will be queued up to for all other clients currently logged in to the server.

A.6 Inter-Client Communication

A.6.1 SESSION-LIST-REQUEST

The SESSION-LIST-REQUEST message asks the server for a list of all the public session ID's of the other clients logged in. This list can be used to send an inter-client message (see below).

Client Request	COMMAND=PROCESS SESSION-ID=*sessionID* DATA-ID=*inputDataID* PARAMETERS=*parameterString*
Server Normal Response	COMMAND=PROCESS-RESULT SESSION-ID=*sessionID* DATA-ID=*resultDataID*

Table A.34: Logical structure of the PROCESS message

The server responds to this message by returning a comma-delimited list of all the current public session ID's of the other clients. Note that the server does not return the requesting client's own public session ID in the list. The comma-delimited list is returned in the SESSIONS field of a SESSION-LIST message.

Client Request	COMMAND=SESSION-LIST-REQUEST SESSION-ID=*sessionID*
Server Normal Response	COMMAND=SESSION-LIST SESSION-ID=*sessionID* SESSIONS=*sessionList*

Table A.35: Logical structure of the SESSION-LIST-REQUEST message

A.6.2 SEND-MESSAGE

This message may be queued up by the server and sent to a client in response to a POLL message.

When sent by a client, this message sends an inter-client message. An inter-client message is a string communicated between two clients logged into the same server. The message is transmitted from the sending client, through the server, and on to the receiving client.

To send a message, a client send the server a SEND-MESSAGE message. The sending client specifies the receiving client with the DESTINATION-SESSION-ID field. This field must contain the public session ID of another client logged in to the server. The MESSAGE field must contain the string to be communicated to the receiving client.

When the server receives a SEND-MESSAGE message, it maps the public session ID sent by the sending client to the private session ID of the receiving client. It also

maps the private session ID of the sending client to its public session ID. Finally, it queues up a SEND-MESSAGE message for the receiving client.

When sent by the server, this message replaces the DESTINATION-SESSION-ID field with the SOURCE-SESSION-ID field. The new field contains the public session ID of the client that sent the inter-client message. Of course, the MESSAGE field still contains the string message to be communicated.

Client	
Request	COMMAND=SEND-MESSAGE SESSION-ID=*privSendSessionID* DESTINATION-SESSION-ID=*pubRecvSessionID* MESSAGE=*messageString*
Server	
Normal Response	COMMAND=ACK SESSION-ID=*sessionID* MESSAGE=SEND-MESSAGE
Queued	COMMAND=SEND-MESSAGE SESSION-ID=*privRecvSessionID* SOURCE-SESSION-ID=*pubSendSessionID* MESSAGE=*messageString*

Table A.36: Logical structure of the SEND-MESSAGE message

A.7 Errors

It is always possible that the server returns an error message in response to any request message. This section describes the four different error messages that the server may return. Note that the SESSION-ID field is never required in an error message.

A.7.1 ERROR

This is the generic error message. It does not specify any specific error. However, an optional DESCRIPTION field may be included to give a text description of the problem.

Server	
Error	COMMAND=ERROR [SESSION-ID=*sessionID*] [DESCRIPTION=*descriptionString*]

Table A.37: Logical structure of the ERROR message

A.7.2 INVALID-SESSION-ID

This error message tells the client that it did not supply a valid session ID in the SESSION-ID field of its request message. This can be caused by two problems. First, the client might have left out the SESSION-ID field when it was required. Second, the client might have provided a session ID that did not correspond to any currently active session. This message will never include a SESSION-ID field. Including one might mislead the client into using the ID again.

Server	
Error	COMMAND=INVALID-SESSION-ID

Table A.38: Logical structure of the INVALID-SESSION-ID message

A.7.3 CONTROL-ERROR

This error indicates that the client tried to change some piece of server hardware state without permission to do so. There is only one client at a time that has control of the hardware. Only that client is allowed to issue certain commands. All such commands are part of the Hardware Control group above.

Server	
Error	COMMAND=COMMAND-ERROR [SESSION-ID=*sessionID*]

Table A.39: Logical structure of the CONTROL-ERROR message

A.7.4 BUSY

The BUSY message is not a fatal error. It simply means that the server cannot handle the client's request at this time. The client should try again later.

This message is most often sent to clients when the server is going through a shutdown cycle (see SHUTDOWN message above).

Server	
Error	COMMAND=BUSY [SESSION-ID=*sessionID*]

Table A.40: Logical structure of the BUSY message

Bibliography

[1] John C. Carney. Message passing tools for software integration. Master's thesis, MIT, June 1995. MTL Memo No. 95-790.

[2] Rick Cattell, Maydene Fisher, and Graham Hamilton. *JDBC Database Access with Java.* Addison-Wesley, July 1997.

[3] Jared D. Cottrell. Distributed data store for MEMS inspection system. Technical report, MIT, May 1998.

[4] Jared D. Cottrell and Erik J. Pedersen. Proposed MEMStation messages. Technical report, MIT, January 1998.

[5] C. Quentin Davis. Measuring nanometer, three-dimensional motions with light microscopy. Technical report, MIT, May 1997.

[6] R. Fielding, UC Irvine, J. Gettys, J. Mogul, DEC, H. Frystyk, T. Berners-Lee, and MIT/LCS. Hypertext transfer protocol—HTTP/1.1. http://www.w3.org/Protocols/rfc2068/rfc2068, January 1997.

[7] David Flanagan. *Java Examples in a Nutshell.* O'Reilly, first edition, September 1997.

[8] David Flanagan. *Java in a Nutshell.* O'Reilly, second edition, May 1997.

[9] Dennis M. Freeman and C. Quentin Davis. Using video microscopy to characterize micromechanics of biological and Man-Made micromechanics (invited). *Technical Digest of the Solid-State Sensor and Actuator Workshop, Hilton Head Island*, pages 161–167, June 1996.

[10] Stanley S. Hong. Instructions on using flick. http://umech.mit.edu/info/eck.html, 1997.

[11] Hughes Technologies. Hughes technologies. http://hughes.com.au/.

[12] James Kao, Somsak Kittipiyakul, and Donald E. Troxel. Internet remote microscope. Technical report, MIT, 1996. CAPAM Memo No. 96-12.

[13] Z. Z. Karu. Fast subpixel registration of 3-D images. Technical report, MIT, September 1997.

[14] Somsak Kittipiyakul. Automated remote microscope for inspection of integrated circuits. Master's thesis, MIT, 1996. CAPAM Memo No. 96-9.

[15] Erik J. Pedersen. User interface for MEMS analysis system. Technical report, MIT, March 1998. Internal MEMS Memo Series No. 98-1.

[16] Manuel Perez. Java remote microscope for collaborative inspection of integrated circuits. Master's thesis, MIT, May 1997.

[17] Ramon L. Rodriguez. Image sampling rate controller. Technical report, MIT, May 1998. Internal MEMS Memo Series No. 98-8.

[18] Ramon L. Rodriguez and Carlos Labrada. Strobe pulse generator for MEMS workstation. Technical report, MIT, December 1997. Internal MEMS Memo Series No. 97-4.

[19] Sun Microsystems, Inc. Java native interface. http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html.

[20] Sun Microsystems, Inc. The java servlet API. http://www.javasoft.com/marketing/collateral/servlets.html.

[21] Sun Microsystems, Inc. Java web server. http://jserv.javasoft.com/products/webserver/index.html.

[22] Sun Microsystems, Inc. Jdk 1.2 Beta 4 documentation. http://java.sun.com/products/jdk/1.2/docs/index.html.