Message Passing Tools for Software Integration

by

John C. Carney

B.S. in Electrical Engineering, Tufts University (1993)

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

© John C. Carney, 1995. All rights reserved.

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

Author .....
Department of Electrical Engineering and Computer Science
April 26, 1995

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

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 17 1995

LIBRARIES

Barker Eng

Message Passing Tools for Software Integration

by

John C. Carney

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

Abstract

As the base of developed software grows, software integration is becoming increasingly important. Many software systems are large and complex. Since re-writing the entire system is not generally possible, newly developed programs must be integrated into the existing system. It is the responsibility of the developer to not only write a new program or tool, but to make it work within the existing environment.

There are several approaches to software integration. Programs within the Unix environment typically are integrated by operating on a common set of files. One program is used to create a file, another to process it, and perhaps yet another to analyze the results. Other approaches include program databases and remote procedure calls (RPCs). An alternative approach for software integration is through the use of a message passing system. The goal of the message passing system is to provide a method by which structured information may be exchanged between two or more running processes. These processes may possibly be running on a single workstation or on multiple workstations connected by a network.

While there are not many message passing systems in existence, the majority which do exist have been designed to be used for a special purpose or in a particular context. There are some general purpose message passing systems, however, these systems impose a fixed integration architecture and message format. A more flexible peer-to-peer approach to message passing has been developed. The peer-to-peer messaging system allows the software developer and integrator to completely design and choose the desired integration architectures and message formats.

The Communications Handling Application Tool Suite (CHAT) has been developed and provides the programmer with a suite of libraries which can be used to integrate software programs using the peer-to-peer message passing approach. There are three libraries of routines, providing support for packaging data into messages, handling incoming messages, and application connection. All three libraries are implemented both in C and Tcl/Tk, a scripting language and toolkit for creating graphical user interfaces under X-Windows. The CHAT suite will allow arbitrary combinations and architectures of C programs and Tcl/Tk programs to exchange messages.

Libraries within the CHAT suite have been used to integrate a new factory display program into CAFE, MIT's Computer Aided Fabrication Environment. CAFE is a software system for the use in the manufacture of integrated circuits, and provides day-to-day support for both research and production facilities at MIT. The factory display program is a graphical program which displays a map of a semiconductor manufacturing facility, including the machines and lots within the facility. The CHAT suite has also been used within other software integration efforts and plans have been made to use the libraries in future integration projects within MIT's Computer Integrated Design and Manufacturing (CIDM) project.

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

Acknowledgments

I arrived at MIT in the fall of 1993 and was appointed to be a teaching assistant in Professor Troxel's digital design laboratory (MIT's famous course 6.111). Professor Troxel's leadership and expertise helped to turn my year as a teaching assistant into one of my greatest learning experiences. When an opportunity to be a part of Professor Troxel's research group became available to me; my decision to join the group was a very easy one to make. There are many people who have helped with my research work over this past year, but without Professor Troxel's support and advice it would not have been possible.

I would like to thank those who have helped me over this past year. Professor Duane Boning has provided excellent advice and support throughout my research work. When my software was under development and documentation was nonexistent, William Moyne served patiently as my first user. He helped me to work out many bugs and I continually bounced ideas off of him. William also sacrificed a night of sleep to help me port my software to OS/2. Asbjoern Bonvik made excellent suggestions which helped me when I was defining the factory display command language. He has also successfully demonstrated an integration of the factory display program with his own facility scheduling software. Greg Fischer was instrumental in the factory display / CAFE integration. His efforts helped to make this integration project run smoothly. Thomas Lohman has helped me with his knowledge of Unix and has helped to isolate and debug a tricky problem within my code.

I would like to thank my officemates James Kao, Jimmy Kwon, Myron Freeman, Michael McIlrath, and Francis Doughty for providing a pleasant and (sometimes too) friendly environment in which to work. It really was pleasure to work among such a great group of people.

I have been incredibly fortunate to have had so much emotional support from my mother, my father, Lea and Julie. They deserve a tremendous amount of credit.

This work has been made possible by support from the Advanced Research Projects Agency (ARPA) contract #N00174-93-C-0035.

Table of Contents

List of Figures 11
List of Tables 13
1 Introduction 15
1.1 Software Integration 16
1.2 Message Passing Systems 17
1.3 A Peer-to-peer Approach 19
1.4 The Communications Handling Application Tools Suite (CHAT) 19
1.5 Exercising the CHAT Suite 23
1.6 Organization of Thesis 23
2 A Message Passing Example 25
2.1 The Electronic Viewgraph 25
2.2 Electronic Viewgraph Interface 26
2.3 Viewgraph Server 28
2.4 Defining the Messages 28
2.5 Defining the Protocols 30
2.6 Summary 34
3 The Connection Support Library 37
3.1 Connection Support 37
3.2 Sockets 38
3.3 Establishing a Connection 38
3.4 C Connection Library Routines 40
3.5 Tcl Connection Library Routines 43
3.6 Other Socket Types 45
3.7 Connection Library Example: The Echo Server and Clients 45
3.8 Summary 49
4 The Communications Handling Library 51
4.1 Handling Newly Connected Clients 52
4.2 Definition of a Message 56
4.3 Standard Message Format and Parsers 57
4.4 Handling of Incoming Messages 60

4.5 Event-Based Message Handling 61
4.5 Queue-Based Message Handling 65
4.7 Combination Message Handling 68
4.8 Summary 70
5 The Message Packaging Library 71
5.1 The World of Message Data Objects 72
5.2 C Library Routines for Message Packaging 73
5.3 Tcl Library Routines for Message Packaging 78
5.4 Summary 80
6 Exercising the Tools 81
6.1 CAFE Overview 81
6.2 Factory Display Overview 81
6.3 Factory Display Integration with CAFE 83
6.4 Sending Messages from CAFE 84
6.5 Communication between fdaemon and fdisplays 84
6.6 Initial State of the Display 84
6.7 Integration Architecture 85
6.8 Facility History and Facility Simulations 85
6.9 Run by Run Control Integration 86
6.10 Other Integration Projects 86
6.11 Summary 87
7 Conclusion 89
7.1 A Graphical Tool for Software Integration 89
7.2 Message Verification 90
7.3 Time-out Mechanism 90
7.4 Efficient Message Formats 91
7.5 Lightweight Version of the Libraries 91
7.6 Porting Libraries to Other Platforms 91
7.7 Summary 92
References 95
Appendix A - Using the Library Source Code 97
The C Source Files 97
The Tcl Source Files 97
Appendix B - Viewgraph Source Code 99
Appendix C - Message Formats 107
The Binary Message Format 107
Object File Header 108
Object File Directory 108
The ASCII Message Format 110
Appendix D - Factory Display Details 113
User's Manual 113
Obtaining Information From the Display 115
Customizations 117
The fdisplay Command Language 118

Socket Communication 123
Outgoing Commands 123
The fdriver Program 125
Input File Format 125
Other Commands 129
The fdaemon Program 130

List of Figures

1 Message Passing via Central Message Server 18
2 Example of Peer-to-peer Message Passing 20
3 Layers of the CHAT Suite 21
4 Viewgraph Windows 27
5 Viewgraph Server Dynamic Model 31
6 Viewgraph Server Substates 32
7 Viewgraph Client Dynamic Model 34
8 Viewgraph Client Substates 35
9 Viewgraph Client Substates 36
10 Connection Flow of Events 39
11 Communications Handling Library 52
12 Example fdisplay Main Window 82
13 CAFE / Factory Display Integration Architecture 85
14 Facility History / Factory Display Integration Architecture 87
15 Binary Message Format 109
16 Populated Main Window 114
17 Hypertext Information Window 116
18 Customize Window 118
19 fdriver Input File Format 126
20 Sample Input File 129

List of Tables

1 Messages Used by the Viewgraph Application 29
2 C Connection Library Routines 41
3 Connection Error Codes 43
4 Tcl Connection Library Routines 43
5 Echo Server C Code 45
6 Echo Client C Code 47
7 Echo Client Tcl Code 48
8 C Library Routines for Handling Newly Connected Clients 54
9 C Skeleton Printer Server Program 55
10 Tcl Library Routines for Handling Newly Connected Clients 55
11 Tcl Skeleton Printer Server Program 56
12 Standard Message Formats 57
13 Example Parser Scenarios 59
14 Example Parser Implementation in C 60
15 Example Parser Implementation in Tcl 60
16 C Library Routines Event-Based Message Handling 63
17 C Library Routines for Event-Based Message Handling 64
18 Example of Event-based Multiuser Echo Server 65
19 C Library Routines for Queue-Based Message Handling 66
20 Tcl Library Routines for Queue-Based Message Handling 67
21 C Library Routines for Combination Message Handling 69
22 Tcl Library Routines for Combination Message Handling 69
23 Data Types 72
24 C Library Routines for Message Packaging 73
25 C Message Packaging Library Example 76
26 Tcl Library Routines for Message Packaging 78
27 C Library Source and Header Files 97
28 Tcl Library Source Files 98
29 Viewgraph Server Source Code 99
30 TIFF Types 109
31 Configuration Prototypes 119

32 Object Placement and Removal Commands 120
33 Object Manipulation Commands 122
34 Utility Commands 122
35 Outgoing Commands 124
36 Allowable Commands in Input File 127

Chapter 1

Introduction

Unix-based environments typically provide a rich abundance of tools. Generally these tools are integrated by operating on a common set of files. One tool is used to create a file, another to process it, and perhaps yet another to analyze the results. There is no interaction amongst tools and each provides its own interface.

A program database is one method used to integrate tools [Rei90]. In this method a single database is used to store all the necessary information about a system. Tools are integrated by having access to a common set of data structures. It can often be difficult to integrate a tool into this type of system. Before a tool can be written, a programmer must have full understanding of the program representation.

An alternative is through the use of remote procedure calls (RPCs). In this model control is passed from a calling application to a procedure which runs within a remote application. The procedure processes any arguments passed to it and returns results and control back to the calling application. While other models may be built using RPCs, by itself RPCs might not provide a rich enough set of features [Dic94].

Another approach for integrating tools is through the use of a message passing system. The goal of a message passing system is to provide a method by which structured information can be exchanged between two or more running processes. These processes may be running on a single workstation or on separate workstations connected by a net-

work. It is the intent that the message passing system will offer, in most cases, an easier alternative for integrating both existing and new tools into a system.

1.1 Software Integration

Software integration can play a very large role in the software development process. There are many instances where integration is both desirable and necessary.

When developing large software systems, development is usually done concurrently, with several programmers each working on their individual pieces of the project. It often provides a good abstraction if many of these pieces are separately running executable programs. For example, when developing a Computer Aided Software Engineering (CASE) environment, the environment can be broken into an editor program, a compiler program, a debugger program, and possibly several other programs [Rei90] [Chu94]. In a highly integrated environment, these programs might interact with each other in some other manner than simply operating on a common set of files. For example, a CASE user might write a program using the editor, and then attempt to compile that program using the compiler. The compiler program might find syntax errors on various lines of the program and alert the user of these errors by directly highlighting these lines within the editor. Perhaps the user could then query each of the highlighted lines to determine the cause of the error. It is the integrated nature of the combined programs which provides a powerful environment for the CASE user.

As systems are being used, users constantly desire new and better tools. Since rewriting the entire system is generally not possible, these new tools must be integrated into the existing system. It is the job of the programmer to not only write the new tool, but to make it work in the existing environment. A particular example of this occurs in the academic world, through the use of research-based software systems. In this situation there is a constant turnover of students, each adding his or her own programs to an existing system. It is usually desirable for the student to spend more of the effort in developing the new programs, and less of the effort in learning the internals of the existing system and integrating the new programs to work within the system.

One potentially large area of software integration is in the use of graphical user interfaces (GUIs). There are many non-graphic legacy systems which could benefit if they

were converted to use GUIs [Lib94]. One approach would be to rewrite sections of the original source code, creating a single executable with graphical capabilities. Another approach is to write a totally separate and stand-alone GUI program. The non-graphic and GUI programs could then be run in conjunction, exchanging the appropriate data and information as to create a transparent system of programs. In fact, this powerful abstraction of separate integrated non-graphic and GUI programs can also be applied when creating new applications, and is not limited to integration with legacy systems. Often the task of integration is forced upon programmers after-the-fact. When a system is designed with modularity and integration in mind, as in the case of integrating a new non-graphic program with a new GUI program, the level of abstraction can lend itself to quicker, easier, and more flexible development.

1.2 Message Passing Systems

The basic idea behind message passing systems is that integrated programs establish connections with each other in some manner. Connected programs can then pass messages among themselves. These messages may contain information which is being made available to connected programs as well as any other kinds of requests or notifications. In general, the message passing system will provide mechanisms for establishing connections, handling of messages, and packaging of data into messages.

While there are not many message passing systems in existence, the majority that do exist are not well known because they are often designed and tailored to be used for a special purpose or in a particular context. There are, however, some general purpose message passing systems which have been developed. Two of these are ToolTalk™, a commercial product developed by SunSoft Inc. [Sun91], and Msg, developed at Brown University as part of Field [Rei90], a highly integrated software development environment.

Both ToolTalk and Msg require tools to register their interest in particular messages with a central message server. This message server runs as a separate process, and simultaneously handles many sessions of connected tools. All messages are routed through the central message server and pattern matched against the registered interests of connected tools. The messages are then selectively broadcast to those tools whose interest

match. See figure 1 for an example of this type of system. Forcing the use of a central message server places several constraints on users of such a system. All messages must be transmitted in two passes, one to the central message server, and one to the connected tools. The central message server must process all incoming messages; and, as a result, message size and bandwidth may become quickly limited as the number of sessions, connected tools and registered interests increase.

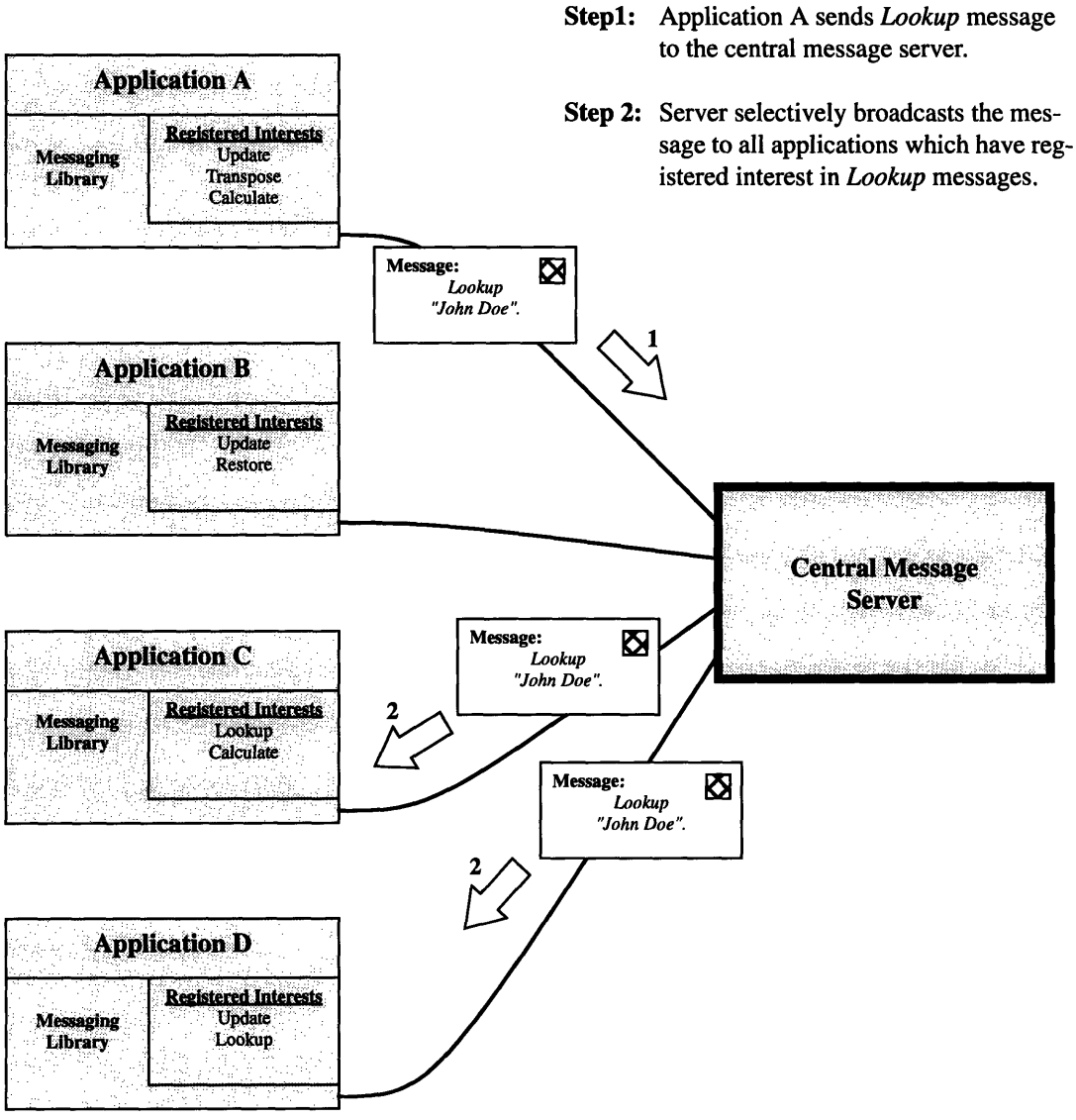


Figure 1: Message Passing via Central Message Server

1.3 A Peer-to-peer Approach

It is clear that as the base of developed software grows, software integration will become increasingly important. Message passing tools can offer an easy solution for many software integration tasks. The central message server based approaches provide a simple architecture for message passing, but have some limitations. Beside the inherent inefficiency in sending messages in two passes, these approaches enforce a standard message format for all connected applications. Programmers and software integrators are not free to design or choose their own message formats. This may especially be a problem when it is desired to integrate software into an existing framework. For example, the Unix *finger* program [Zim91] consists of a daemon program which is always running and client programs. These client programs connect to the daemon, send a message querying for information about the users of the particular machine, and then wait for a reply message from the daemon. If it were desired to write a new *finger* client program, perhaps a graphical one, then the new client would have to communicate using messages of the format which is already being used by the widely installed base of *finger* daemon programs.

A more flexible peer-to-peer approach to message passing has been developed. This approach allows individual programs to establish one or many connections with various other programs. Programs may choose to handle messages differently on various connections, and are not limited to a single message format. In fact, a program might receive messages over a connection of one format, but send messages over that same connection of a different format. The peer-to-peer messaging system allows the programmer and software integrator to completely design and choose whatever connection architectures they desire. If it is desired to use a central message server architecture, this can easily be built using the peer-to-peer messaging system, as these architectures are a subset of those which can be built. An example of peer-to-peer message passing is shown in figure 2.

1.4 The Communications Handling Application Tools Suite (CHAT)

The Communications Handling Application Tools Suite (CHAT) has been developed and provides the programmer with a suite of libraries which can be used to integrate software programs using the peer-to-peer message passing approach. There are three libraries of routines, providing support for packaging data into messages, handling incoming messages and application connection. All three libraries are implemented both in C and in Tcl/

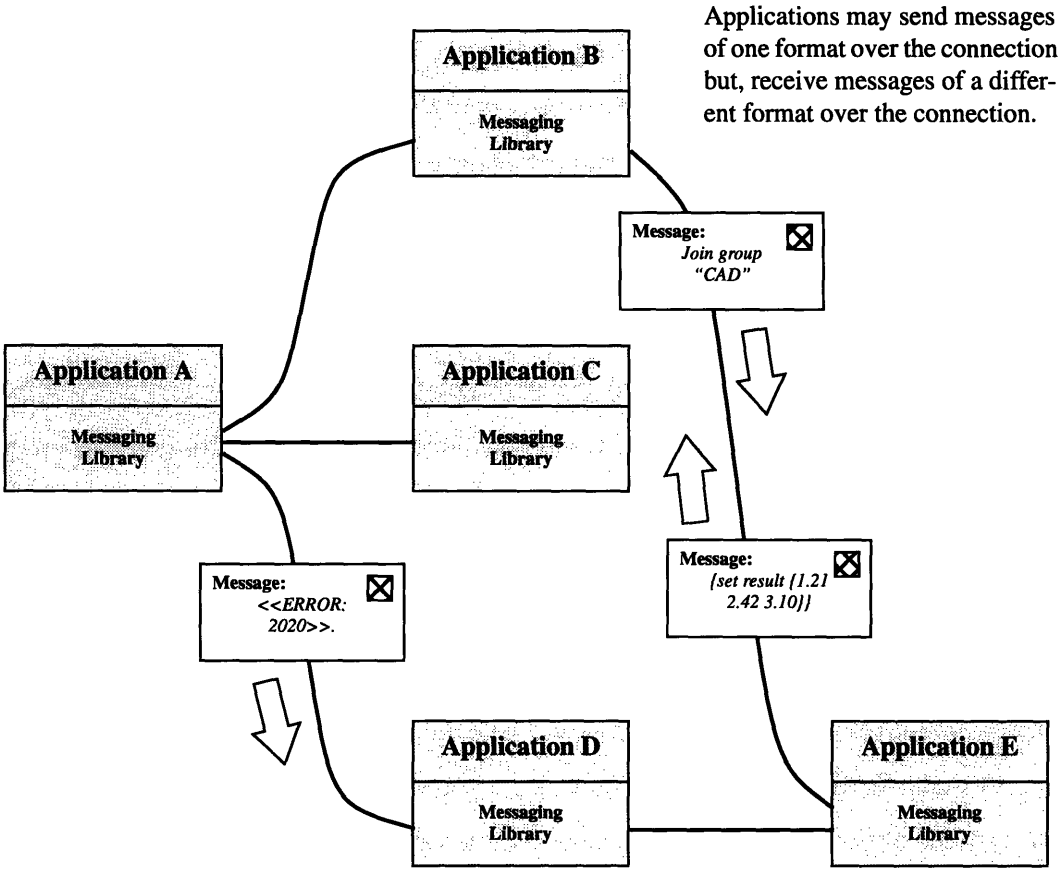


Figure 2: Example of Peer-to-peer Message Passing

Tk, a scripting language and toolkit for creating graphical interfaces under X-Windows [Ous94]. The CHAT suite will allow arbitrary combinations and architectures of C programs and Tcl/Tk programs to exchange messages. The various layers of the CHAT suite are shown in figure 3.

1.4.1 Message Packaging Library

Data may only be sent between connected programs as a contiguous series of bytes. As a result all message formats must built around this constraint. For many integration tasks, simply an ASCII string of text followed by a newline character might suffice as the message format. A program wishing to package more complicated data of varying types and lengths into a single message has a much more difficult job of serializing the data into a standard message format. When the message is received by the connected program, the

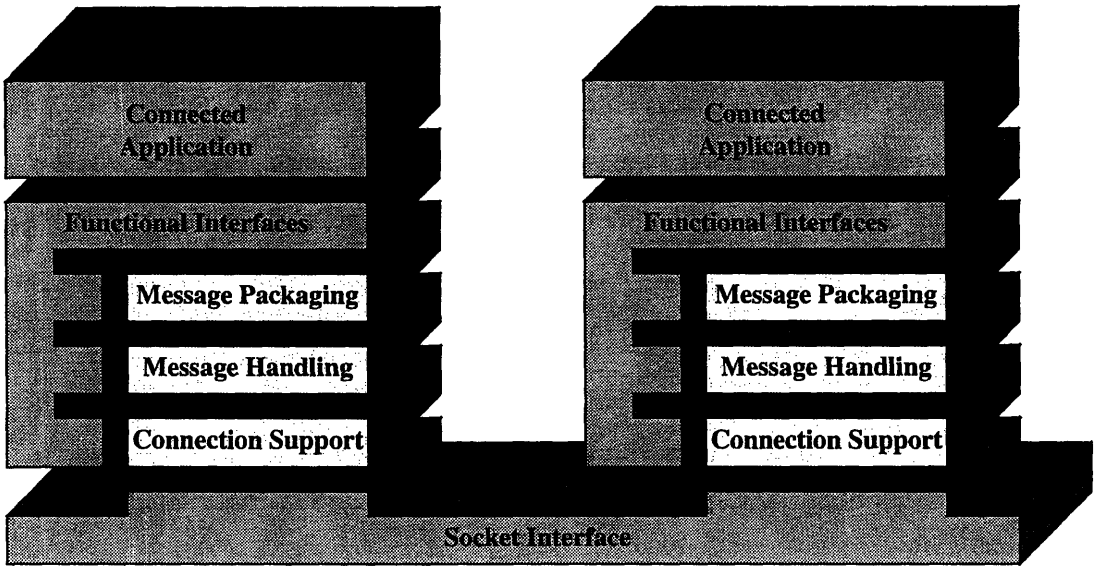


Figure 3: Layers of the CHAT Suite

data must then be extracted into a usable representation. The message packaging library allows a program to easily create a message data object. This object can then be filled with an arbitrary number of slots, where each slot contains a single or a vector of primitive types. The library provides an interface which allows slots to be easily set and queried. When all of the appropriate data has been packaged into the message data object, a library routine can then transform the internal object into a formatted serial buffer of bytes. This buffer can then be transmitted (possibly over a network) to the connected program. Conversely, when the formatted serial buffer is received by the connected program, a library routine can transform it back into an internal message data object whose data can then be easily queried by the program.

1.4.2 Message Handling Library

When incoming messages are received by a connected program, there are many possible ways to handle the newly arrived message. The message handling library provides a facility for event-based handling, whereby incoming messages on a connection can be bound to an internal handler routine. Messages will then be transparently received by the program. Whenever a complete message arrives, the program will be interrupted and the han-

dler routine will be called on the newly arrived message. The message handling library also provides a facility for queue-based handling. Again, messages will transparently be received by the program. Whenever a complete message arrives, it will automatically be placed in the message queue associated with the connection. At any time the program may check the status of the message queue, and pop messages from the queue as desired. The message handling library also provides a third method of message handling which is a combination of event-based and queue-based handling. In this method, a program can toggle message handling on a connection between event-based and queue-based whenever desired.

The message handling library is at the core of the CHAT suite. It has been designed with flexibility in mind and places very few constraints on the message formats which may be handled. As a result, the formatted serial buffers generated by the message packaging library can be handled by the message handling library. However, if the integration task is not compatible with that format, then practically any other message format can be used. This may occur, for example, when a new program is to be integrated into an existing framework or in the case when the integration task requires only a very simple message format and not the complete capabilities of the message packaging library.

1.4.3 Connection Support Library

This library of routines provides a simple interface to Berkeley Sockets and the TCP/IP protocols. The TCP/IP protocols are used by the Internet (and many other networks) for network communication. The Unix operating system provides sockets as an interface to the TCP/IP protocols as well as a number of other protocols. The connection support library restricts use to two kinds of sockets. These are TCP sockets and Unix Domain sockets. TCP sockets use the TCP/IP protocols to establish a communication stream between any two programs which are running on the same system, or between any two programs running on different machines and connected by the Internet. Unix Domain sockets can only be used to establish a communication stream between any two programs running on the same system. It is not necessary to provide support for Unix Domain sockets, but when it is known that two programs will be running on the same system, Unix Domain sockets will deliver more efficiency. By restricting use to TCP sockets and Unix

Domain sockets, the connection support routines are much cleaner and easier to use than the direct socket system calls provided by the Unix operating systems. Since these two types of sockets represent the great majority of sockets currently in use, there is virtually no loss of flexibility when integrating new programs into existing software systems.

1.5 Exercising the CHAT Suite

Libraries within the CHAT suite have been used to integrate a new factory display program into CAFE, MIT's Computer Aided Fabrication Environment. CAFE is a software system for use in the manufacture of integrated circuits, and provides day-to-day support for both research and production facilities at MIT [McI92]. The factory display program is a graphical program which displays a map of the semiconductor manufacturing facility, including the machines and lots within the facility. CAFE uses massage passing to communicate with a factory display daemon. Multiple factory display programs can simultaneously be connected to the factory display daemon and also communicate with it by using message passing. As the status of machines and lots change within the CAFE system, messages are sent to the various displays which are then be updated to reflect the new status information.

Libraries within the CHAT suite have also been used to integrate a graphical user interface and simulation environment with a run by run control server [Moy95]. Plans have been made to use the CHAT suite in research work being done on the remote fabrication and remote inspection projects within MIT's Computer Integrated Design and Manufacturing (CIDM) project.

1.6 Organization of Thesis

This chapter has served to present a background of software integration and message passing tools as well as an overview of new work which has been done in creating peer-to-peer based message passing tools for software integration, including a description of the CHAT suite. Finally, some examples of how these tools have been exercised were outlined.

In the following chapter a specific message passing application will be discussed and analyzed. Then the libraries within the CHAT suite will be further presented and described in detail. The factory display integration into CAFE, which was introduced earlier in this chapter will be fully discussed. Finally, some limitations and ideas for future

work will be presented. Wherever possible, detailed descriptions and code examples will be preceded by sections of overview, so that readers may selectively skip the details as desired and re-visit them as it is necessary.

Chapter 2

A Message Passing Example

The message passing tools within the CHAT suite provide a great deal of functionality and flexibility. Before presenting the libraries themselves, an example application which uses message passing is discussed here. It is the intended that this example will provide a higher level starting point for understanding message passing, without delving into the code or routines which are used to implement such an application. The code is provided in the appendix. The example presented here is specific, however, an understanding of the message passing nature of the example will serve as a skeleton on which to build a much broader range of applications.

2.1 The Electronic Viewgraph

When people, both in industry and in the academic world, schedule a meeting to present new information they very often make use of a viewgraph projector. This allows the speaker to project pictures onto a screen, which he or she can then control and point to throughout the presentation. Communication among long-distance colleagues is often done with a telephone conversation. These telephone conversations are either between two individuals, or an entire group of participants by means of a conference call. The idea of the electronic viewgraph, is to allow the speaker to project electronic images from his or her computer onto the computer screens of the other participants, provided that the computers of the participants are connected by a network. The speaker can then use a pointer

to point on the local image, while the long-distance participants look at the same image and pointer on their own computer screens. The speaker controls which pictures are displayed, when to display them, and where the pointer is pointing. The other participants in the viewgraph session are locked-out from having any control of the viewgraph, however, speakers may turn over their control as they desire. The electronic viewgraph provides a nice interface and is as simple to use as its non-electronic counterpart.

2.2 Electronic Viewgraph Interface

It is easy to see that message passing might be useful as a communication mechanism among electronic viewgraph programs. For example, when the speaker points to something on his or her local screen, messages might be sent to all other participants' viewgraph programs, supplying them with information about the pointer's state and location. However, before diving straight into message passing details, it will be useful to take a close look at the viewgraph interface and define the program from the user's perspective.

2.2.1 Viewgraph Windows

When a participant of the viewgraph session starts-up the viewgraph program they will get two windows on their screen. One window contains several control buttons, and the other contains a blank canvas region. These are shown in figure 4. The window containing the blank canvas region is the actual viewgraph screen where the pictures and pointer will be displayed during a presentation. The controls window contains buttons which are used to control the viewgraph display. This window also contains a few status labels, providing information about the current state of the viewgraph session.

2.2.2 Program States

From the perspective of a single participant of the viewgraph session there are three states which their viewgraph program may be. Regardless of the state of the program, any participant has the ability to quit the program and exit from the session at any time by simply clicking on the *Quit* button in the controls window. Also, any participant may at any time submit electronic images to the pool of images available to the viewgraph controller. To submit an image, the participant clicks on the *Submit Image* button and then is presented with a window where they can browse their local directories and select a file containing an image. For the sake of simplicity, only image files in the GIF file format are supported by

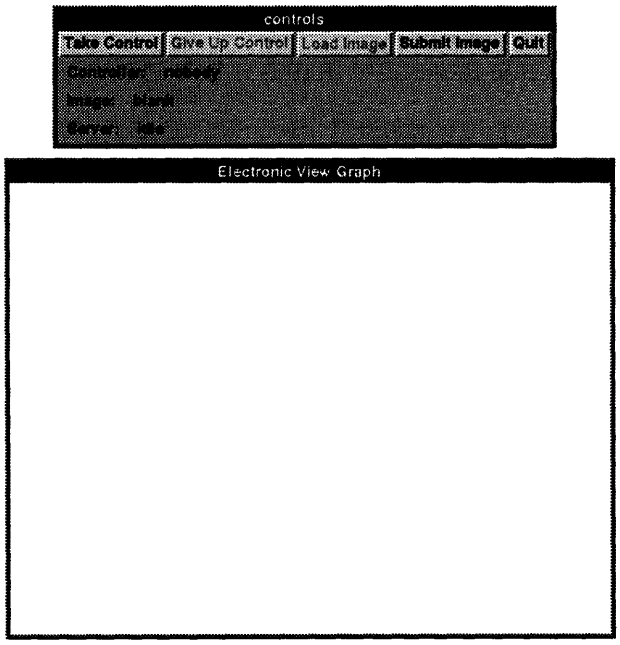


Figure 4: Viewgraph Windows

the viewgraph program.

The first state, is that where no participant has control of the viewgraph. When the program is in this state, the *Controller* label in the controls window will be followed by *nobody* to indicate this. At this point, control is up-for-grabs. Beside the *Quit* and *Submit Image* buttons only one other button will be active, which is the *Take Control* button. The first participant to click on this button will be given control of the viewgraph.

The second state, is that where the participant owns control of the viewgraph. As controller of the viewgraph, the controller's username and hostname will appear next to the *Controller* label in the controls windows of all the participant. At this point, the *Take Control* button will be disabled, and the *Give Up Control* and *Load Image* buttons will be active. The participant who owns control may at any time click on the *Give Up Control* button. This will then put the viewgraph back into the "nobody has control" state and control is once again up-for-grabs. However, if instead the controller clicks on the *Load Image* button, he or she will be presented with a menu containing the names of all the submitted images. The controller may then choose any image to be displayed. After this, the selected image will appear within the canvas region of all participants' viewgraph windows. The name of the image will appear next to the *Image* label in each participants' con-

trols window. The controller may then point his or her mouse on the displayed image and press the mouse button. While the mouse button is held down, a fixed pointer will appear at the same location on each participants' viewgraph window. When the mouse button is released, the pointer will disappear from all participants windows. The controller may freely load any images and point on them as desired.

The final state, from the point of view of a participant, is that where another participant owns control of the viewgraph. At this point the three buttons *Take Control*, *Give Up Control*, and *Load Image* buttons will all be disabled. The name of the current controller along with the name of the currently displayed image will appear in all participants' controls windows. When in this state, the participant may contentedly watch the presentation and wait for the controller to give up control. Of course, the participant may also submit images or quit from the session at any time.

2.3 Viewgraph Server

It is clear that there must be a high level of coordination amongst all participating viewgraph programs within a session. It is the job of the viewgraph server program to handle this coordination. The idea is that when a new viewgraph session is desired, the viewgraph server program is started and runs for the duration of the entire session. Then each participant starts-up their local viewgraph client, which will connect to the viewgraph server. Clients may join in or leave a session at any time. Each client has the ability to communicate with the server by exchanging messages with it. Some messages will contain images and be very large. When the server is busy handling such messages, *busy* will appear next to *Server* in each clients' controls window, otherwise *idle* will appear. This status indication is to simply let the participants know that the server is busy working and may be sluggish at the moment in responding to other messages.

2.4 Defining the Messages

When designing integrated software, such as the viewgraph application, it is helpful to define the messages which will be used in the communication. Since the viewgraph application consists of two programs, a client and a server, it is useful to think about which messages will be received and sent by the server, and which messages will be received and sent by the client. Complicating this process is the fact that a server and multiple clients

may be running concurrently. For example, in the viewgraph program, when nobody has control of the viewgraph each participant will have an active *Take Control* button. Clicking on this button may result in a message being sent to the server. Since the sending and handling of the message will take some small amount of time, it is certainly possible that two or more participants will click on this button near simultaneously before the server can respond. The client programs at this point should not assume that they own control of the viewgraph, instead they should only assume control of the viewgraph when they receive a message from the server saying that they own control. The defined messages and protocols should unambiguously handle these kinds of subtleties and applications should not be designed on the assumption that the probabilities of such events happening are very small. It will be much easier to design in robustness from the outset, than to fix these problems later.

There are a total of ten messages used by the viewgraph application. The following table lists these messages along with a brief description and overview including the information contained within the message.

Message	Description
PRESUBMIT	This message contains no data. It is sent by a client to simply tell the server that it is about to send a *SUBMIT* message. The server then sends a *SERVSTAT* message all clients informingthat the server is busy.
SUBMIT	This message contains the name and data for an image. It is sent by a client when it wishes to have the contained image added to the pool of available images.
TAKE	This message is sent by a client when the client desires to take control of the viewgraph. The message contains the username, hostname, and process ID associated with the client program.
GIVE	This message contains no data and is sent by the controlling client when the client desires to give up control of the viewgraph.
DISPLAY	This message is sent by the controlling client and contains the name of the image which the client desires to be displayed. This message is also sent by the server to all clients.
POINT	The controlling client sends this message to the server. The message contains the state and possibly coordinate of the pointer. The server sends this message to all clients.
CONTROL	This message is sent by the server to all clients and contains the username. hostname, and process ID of the controlling client.

Table 1: Messages Used by the Viewgraph Application

Message	Description
PICTURE	This message contains the name and data of an image. The server sends this message to all clients except the client which submitted the image.
SERVSTAT	This message is sent by the server to all clients and contains the status of the server.

Table 1: Messages Used by the Viewgraph Application

2.5 Defining the Protocols

While it is necessary to define the messages which will be exchanged by integrated programs, the messages do not stand on their own. It is also necessary to define the actions which lead up to the messages to being sent, as well as actions which follow when a message is received. This can generally be called the messaging protocol. A nice way to organize the protocol is by creating a dynamic model of each program's states and events. Creating an integrated application is an iterative process, and when designing the dynamic model it may be necessary to go back and add, eliminate, or re-design the actual messages which will be used to communicate among the programs. Figure 5 shows the dynamic model for the viewgraph server. The dynamic modelling concepts and notations used are those proposed by James Rumbaugh [Rum91]. The dynamic model contains several states, each of which have one or many substates. Figure 6 shows the viewgraph server's substates.

The viewgraph server is event-driven. The program spends the great majority of its time in an idle state and only acts when an event occurs. This event may be a new client connecting, a client death, or a newly arrived message. When an event occurs the server will enter a state where it handles the event. It is certainly possible while in the process of handling one of these events that another event will occur. The dynamic model presented doesn't allow for this possibility and restricts these events to only occur when the server is in the idle state. As a result, the message passing tools must provide a mechanism for accomplishing this desired behavior. Once the server starts running, there is no specification in the dynamic model to end the server program. The program must be explicitly killed when the viewgraph session is over.

It is the job of the viewgraph server to store all of the images which have been submitted by the clients. This is necessary to allow for new clients to connect into the session

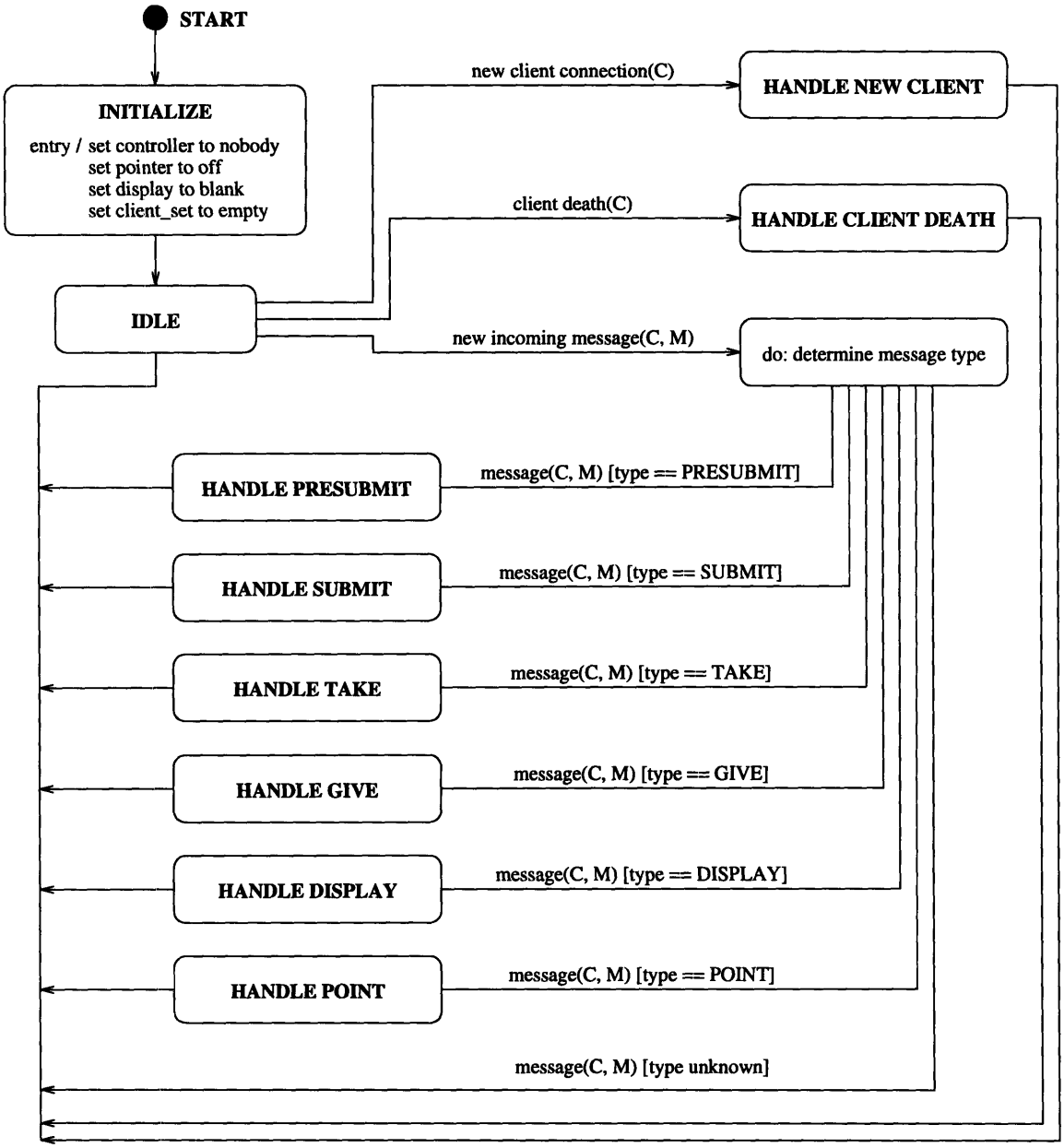


Figure 5: Viewgraph Server Dynamic Model

at any time. The *HANDLE NEW CLIENT* substate diagram shows that all saved images are sent to newly connected clients upon initial connection to the server. The idea is that each client will cache these images locally, so that when the viewgraph controller selects an image to load, a short message is sent telling each client which image to load, instead of a large message being sent containing the image itself. The *HANDLE SUBMIT* substate diagram shows that whenever a new image is submitted to the server, it is saved by the

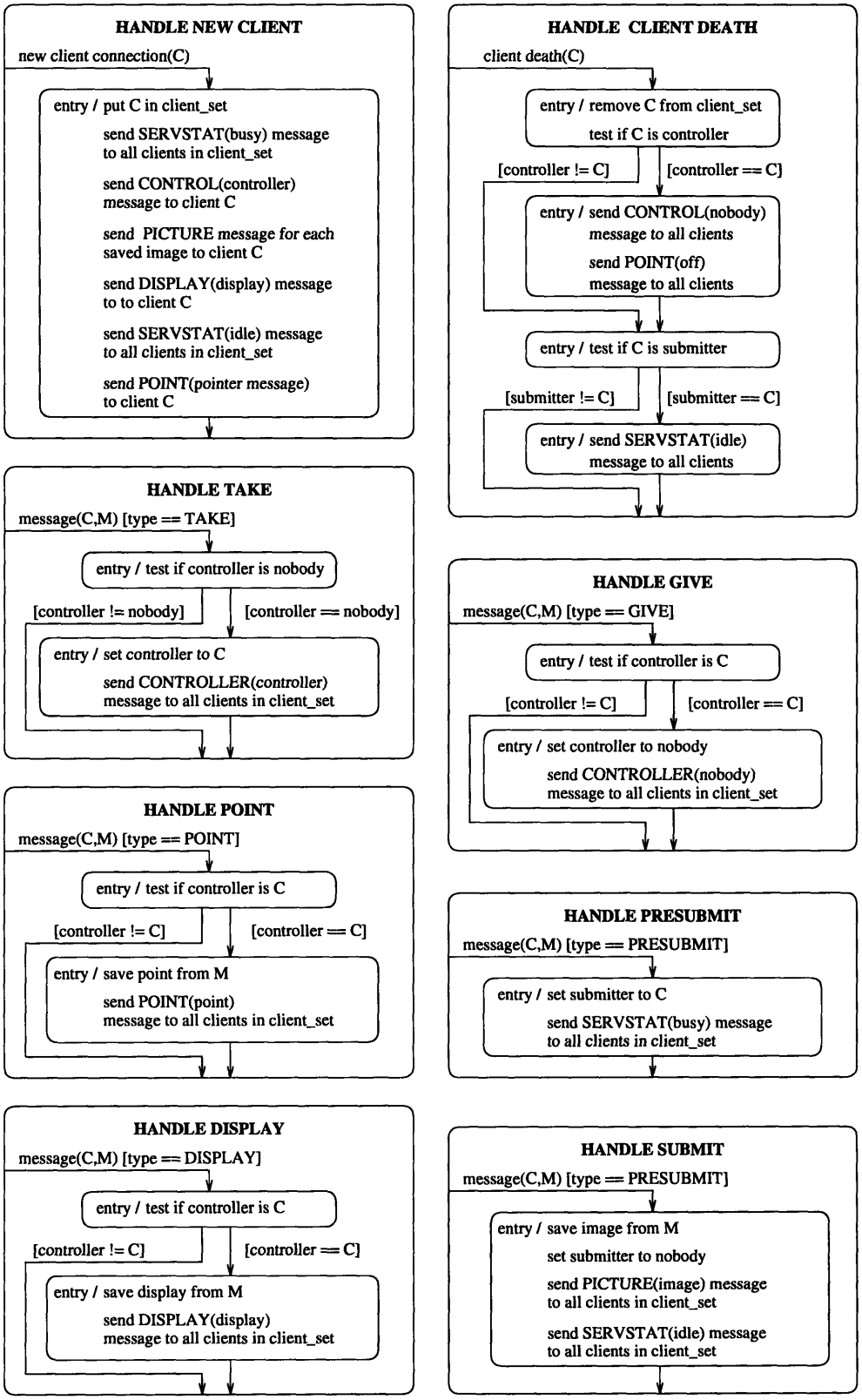


Figure 6: Viewgraph Server Substates

server and then immediately sent to all of the connected clients.

One thing to note about the protocol is that the *POINT*, *DISPLAY*, and *GIVE* messages should only be sent by the client who owns control of the viewgraph and never sent by a non-controlling client. However, when the server receives one of these messages it checks for this condition and if it finds that the message was not sent by the controlling client, then it will simply ignore the message. Likewise, the *TAKE* message should only be received by the server when no client has control. The server also checks for this condition and will ignore the message if there already exists a client which owns the control of the viewgraph. While it is not necessary to check these conditions when things are properly implemented, however doing so provides some level of safety for the server against potentially harmful clients which do not follow the specified protocol.

The viewgraph client program is a graphical program and is also event-driven. Beside handling newly arrived messages, the client program must also handle the button clicks, and menu selections of the user. Figure 7 shows the dynamic model for the viewgraph client program. The program spends most of its time in the idle state, with events handled as they occur. Although certain events would cause the program to leave the idle state such as a button click, these events can only occur when the button is enabled. Care must be taken when interpreting the dynamic model to understand the various conditions placed on these events. The substates of the viewgraph client program are shown in figure 8 and in figure 9.

Upon start-up the viewgraph client's buttons are disabled. These buttons then become enabled and disabled based on the actions of the local user and the other remote users. One thing to note about the client program is that upon the user clicking the *Quit* button, the program simply cleans up locally and exits. No message is sent to the server informing it that the client is about to exit. The server already has the ability to handle an unexpected client death. In the case of the viewgraph program unexpected and expected client exits are handled in exactly the same manner by the server. An additional *EXITING* message is not a necessary part of the protocol. However, if the application required that unexpected and expected client deaths be handled differently, then it would be a good idea to include an extra message in the protocol for this purpose.

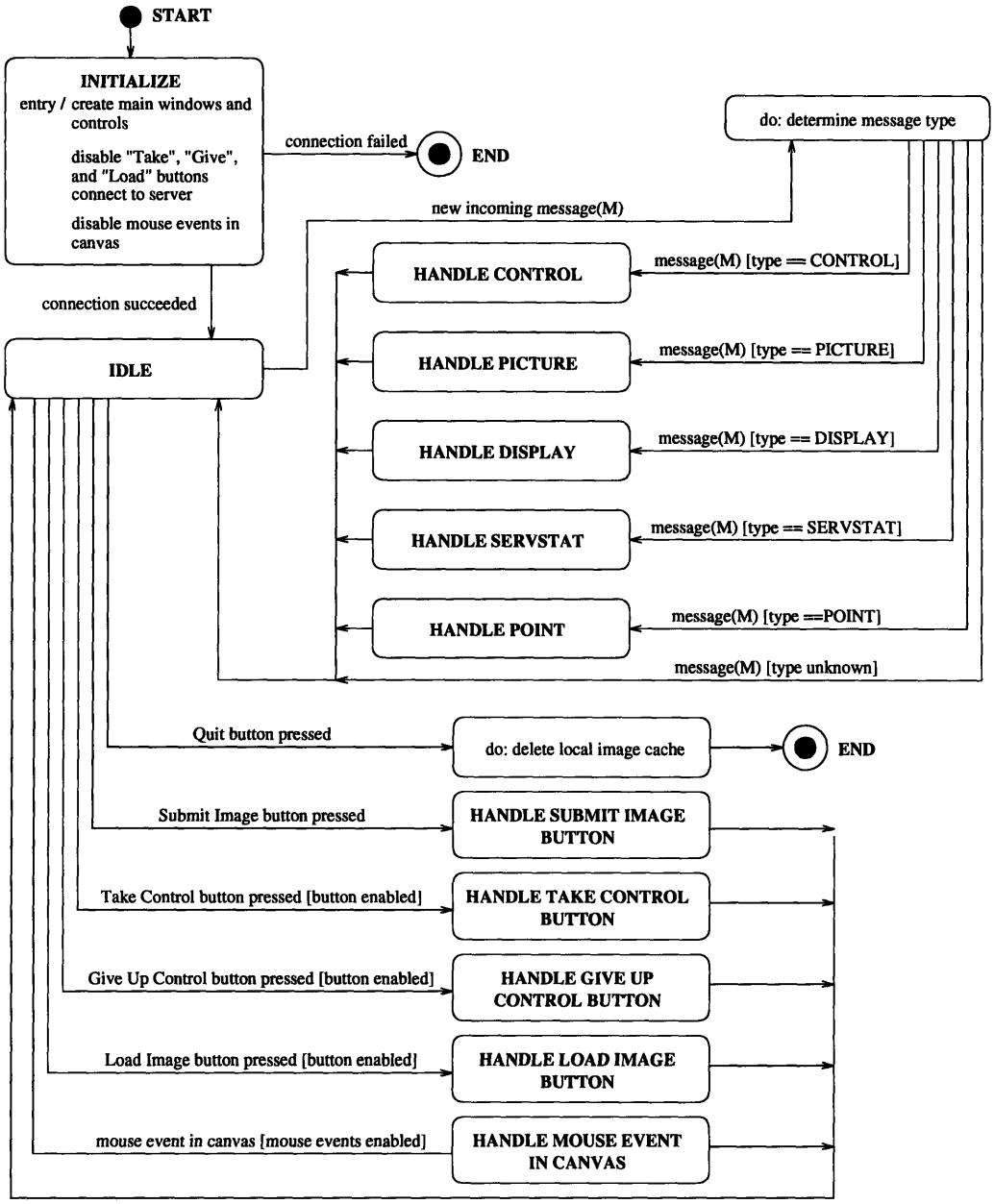


Figure 7: Viewgraph Client Dynamic Model

2.6 Summary

This chapter has introduced the electronic viewgraph application. It has presented a model of a solution for the integration and message passing which may be used in an implementation of such a program. During any viewgraph session one server program and many client programs may be connected and running. The programs have a set of ten messages which get exchanged among the clients and servers as the participants in the viewgraph session act upon their own graphical interfaces. The dynamic models attempt to show the

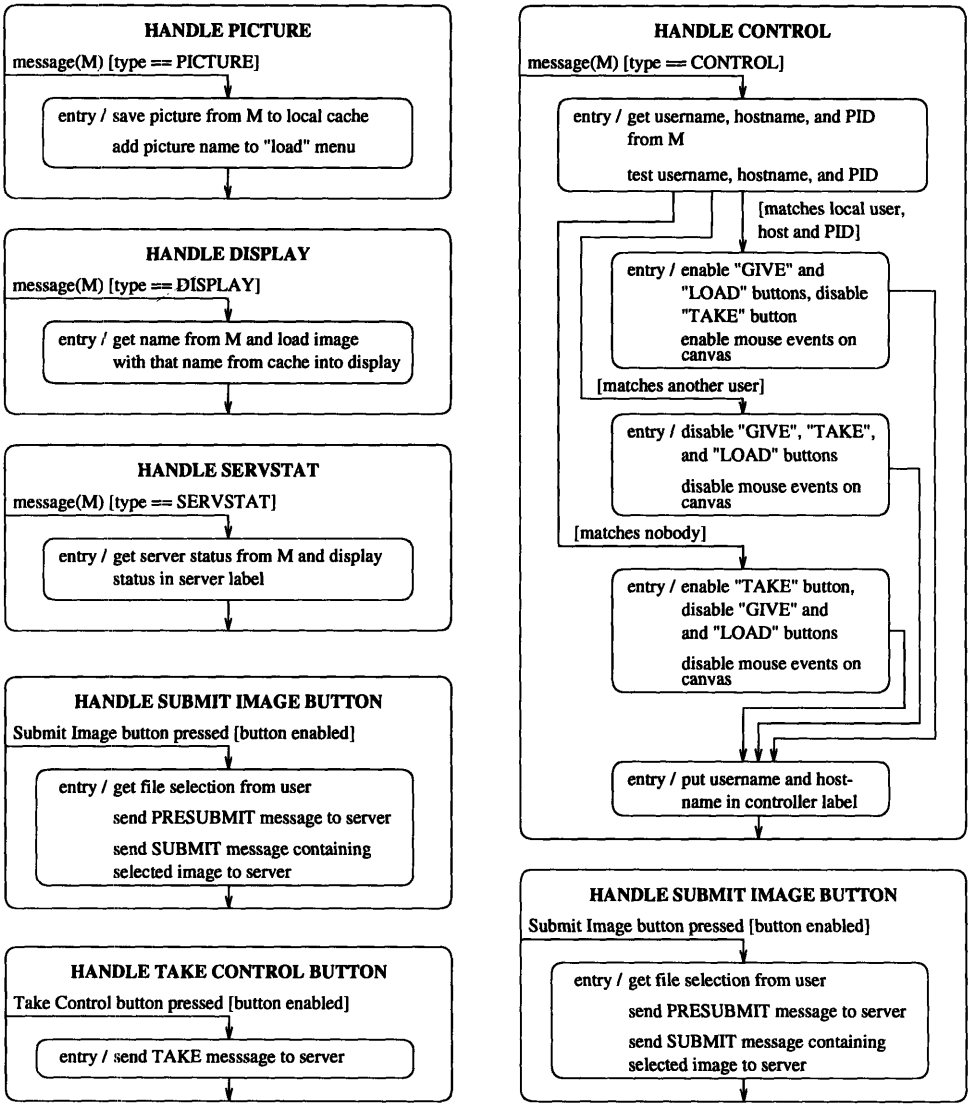


Figure 8: Viewgraph Client Substates

actions which lead up to a message being sent, as well as the programs' responses to the messages when they are received.

While it is easy to understand the purpose and interface of the viewgraph application, the message definitions and protocols are fairly complex. The modelled solution has been presented with hardly any regard to the tools which will be used to actually connect the programs, package the messages, and handle the received messages. The difficult part of the integration process is in designing the messages and protocols, for each individual problem will lend itself to a unique design. It is intended that the flexible tools provided by

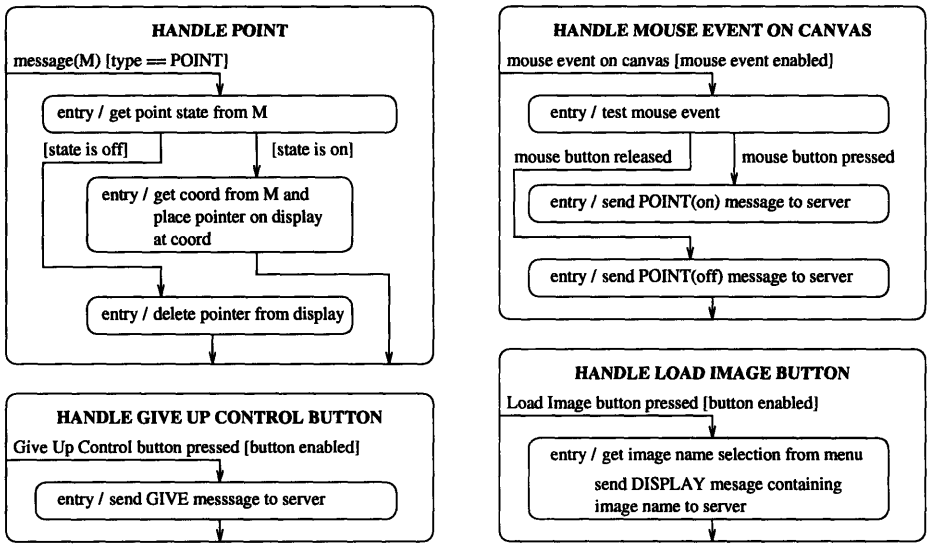


Figure 9: Viewgraph Client Substates

the CHAT suite will allow the programmer to concentrate their efforts on designing solutions to the difficult aspects of the integration problem without having to be overly concerned about or constrained by the details of the underlying network connections, message formats, or message handling mechanisms.

Chapter 3

The Connection Support Library

Before messages can be sent among connected programs, these programs must be connected to each other. The goal of the connection support library is to provide a simple programming interface to network communication.

3.1 Connection Support

The connection support library allows two running programs on the same machine or on different machines connected by a network to establish a full duplex communication channel over which data may be transferred. The library uses the Berkeley socket interface provided by the Unix operating system as an interface to the TCP/IP protocols. Sockets provide a level of abstraction which allows network I/O to be similar to file I/O. However, there are many more details and options involved with network I/O [Ste90], including addressing, as well as the wide range of communication protocols supported. By restricting the connection support library to two type of sockets, TCP sockets and Unix domain sockets, network communication and connection establishment can be greatly simplified. TCP sockets are used to establish reliable stream connections between two processes on different systems or two processes on a single system. TCP sockets use the underlying TCP/IP protocols, which are the protocols used by the Internet. As a result connections may be established between two programs connected by the Internet. Unix domain sockets may only be used to establish reliable stream connections between two processes on the

same system. While the use of Unix domain sockets is not necessary, they do provide more efficiency and should be used when it is known that two connected processes will be running on the same system.

3.2 Sockets

A socket is defined as an endpoint for communication. For communication to be possible between two processes, each must have access to a socket in the form of a socket descriptor. The socket descriptor is a handle to the communication channel, and it is used as a reference whenever it is desired to act upon that channel, such as when a program wishes to write data to it or read data from it. Since a socket is only an endpoint for communication, it must be associated with another socket (the other endpoint) in order for data to be exchanged over the communication channel. The routines provided by the connection support library allow a program to create sockets and build the necessary association. Once this association is established, then each program can write to its socket descriptor in a similar manner to writing to a file descriptor. The information written to the socket descriptor can then be obtained by the connected program by reading from its own associated socket descriptor.

3.3 Establishing a Connection

In order for a connection to be established there is a flow of events which must take place The connection process is not symmetric for each program. One program initiates the connection, while the other program listens for and accepts the connection. For both TCP and Unix domain sockets there exists a port (or path) which uniquely identifies where the listening program is listening. This port (or path) is known to both programs beforehand and has an analogous meaning as a telephone number. Establishing a connection between two programs is similar to establishing a connection between two telephones.

3.3.1 Flow of Events

The program which accepts a connection will be called the acceptor, while the program which initiates the connection will be called the initiator. Each program must follow a distinct flow of routine calls in order for the programs to successfully connect and exchange information. There are two scenarios for these flow of events shown in figure 10.

In each scenario there is no restriction on which program starts first, the acceptor

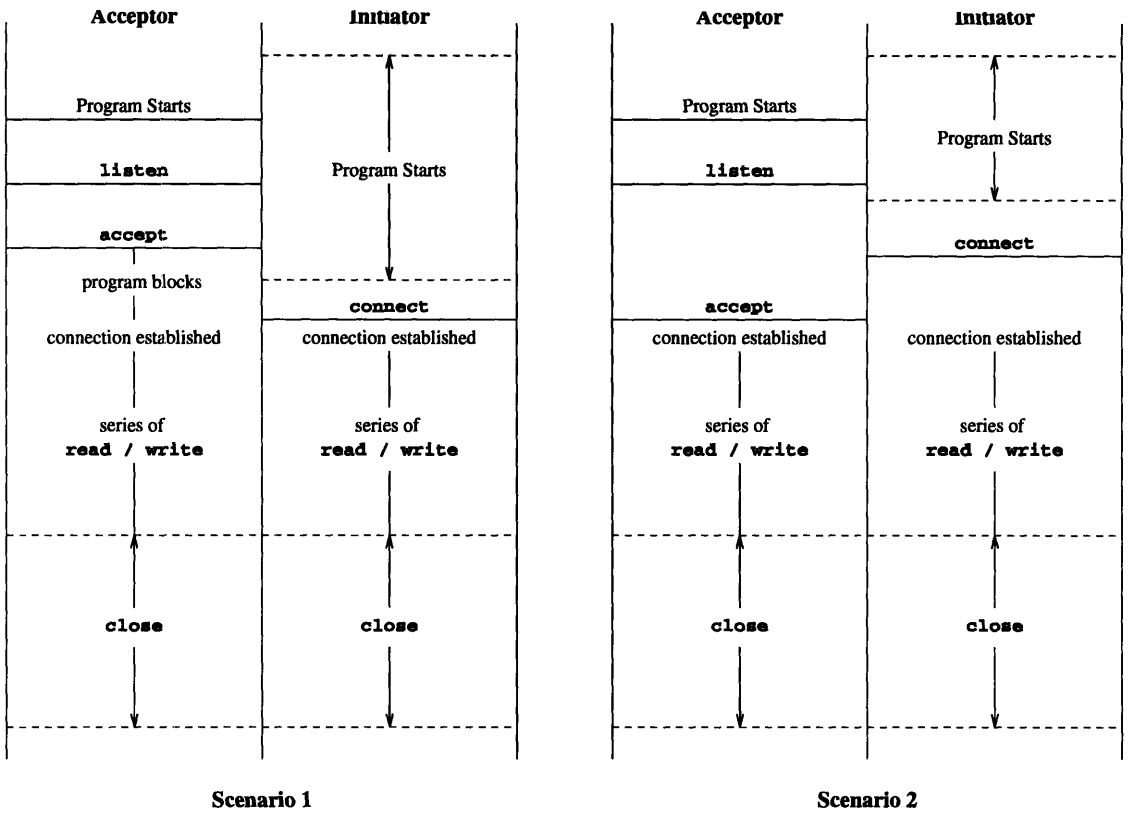


Figure 10: Connection Flow of Events

or the initiator. Before any other connection routines are called, the acceptor must call the *listen* routine. By calling this routine the acceptor program establishes the port (or path) on which it would like the initiator program to connect to. Using the telephone analogy, this is essentially the acceptor program's way of assigning itself a telephone number.

In the first scenario, the acceptor program then calls the *accept* routine. At this point the program desires that an initiator connect to it. The acceptor program will block until this happens. Finally the initiator program issues the *connect* routine and the connection is established. Once the connection is established both programs may issue a series of *read* and *write* calls to exchange information. When the programs have finished, they then close the connection by both issuing the *close* routine.

In the second scenario, the initiator issues a *connect* call before the acceptor issues the *accept* call. In this case, the *accept* routine does not block and the connection is immediately established when the acceptor *accepts*. Again, after the connection is established each program may exchange information with *read* and *write* calls, and then *close* the connection.

3.3.2 Multiple Connections

It is possible for the acceptor program to accept many connections on an port (or path) from one or many initiator programs. To do this the flow of events is simply extended. The acceptor program first calls *listen*. Then the acceptor calls *accept* multiple times throughout the program. For each call to *accept* there is an initiator's call to *connect*. The acceptor program will block if it *accepts* before the corresponding *connect* call, but won't block if it calls *accept* after the corresponding *connect* call.

3.3.3 TCP Connections

In order for a TCP connection to be established, the acceptor program must listen to a port on the local system. A port is simply a 16-bit value, where ports 1-1023 are reserved only for superuser processes. The IP address of the local host machine of the acceptor program combined with the port number forms a unique logical location on the network where the program listens for connections. When an initiator then connects, it will connect to that address and port number.

3.3.4 Unix Domain Connections

Unix domain connections can only be established between two programs on the same machine. In this case the acceptor program listens to a path. The path is the unix domain path of a unique filename. Both acceptor and initiator should have read and write access to the directory where this unique filename exists. A typical place is the */tmp* directory. The initiator program the connects to that unique filename in order establish the connection.

3.4 C Connection Library Routines

The following table lists each routine in the C connection library, along with a description of how the routine is used. To use these routines, the programmer must link in the compiled object code of the library as well as including the appropriate header file in with their

own source code.

`int bs_tcp_listen(port)`
`int port;`
This routine listens to a TCP port on the local machine for connections. The ***port*** argument should be a positive integer where 1-1023 are reserved for superuser processes. Upon success the routine will return a socket descriptor, which is simply a small positive integer. This socket descriptor is a handle to the listening port and will be used as an argument to other routines. Upon failure the routine returns -1.

`int bs_unix_listen(path)`
`char *path;`
This routine listens to a Unix path on the local machine for connections. The ***path*** argument is should be a pointer to a character buffer containing a Unix domain path to a local file. This file should not exist on the system before this call is issued, as the call will cause the operating system to create a special zero sized socket file on the local system. The program which issues this routine as well as programs wishing to connect to it should all have read and write access to the directory where this special file exists. The routine will return a socket descriptor, upon success and -1 upon failure.

`int bs_accept(sd)`
`int sd;`
This routine accepts a connection on a listening socket descriptor. The ***sd*** argument is the listening socket descriptor which was obtained by a call to **`bs_tcp_listen`** or **`bs_unix_listen`**. This routine will block if the connecting client has not yet attempted to connect, otherwise it will return immediately. The routine returns a new socket descriptor which is a handle to the connected client, and can be used as an argument to **`read`** and **`write`** to exchange information with the client. It is possible to accept multiple connections on the same listening socket descriptor, by calling **`bs_accept`** multiple times with the same argument. For each connecting client, **`bs_accept`** will return a new socket descriptor which is connected only with one client. The **`bs_accept`** routine will return -1 upon failure.

`int bs_tcp_connect(host, port)`
`char *host;`
`int port;`
This routine connects to a listening TCP port on a host. The ***host*** argument is a pointer to a character buffer containing the IP address of the host either as a hostname (i.e. "garcon.mit.edu") or in dotted decimal notation (i.e. "18.62.0.242"). Upon successful connection the routine will return a socket descriptor, which can be used as an argument to **`read`** and **`write`** to exchange information with the connected host program. Upon failure the routine will return -1.

`int bs_unix_connect(path)`
`char *path;`
This routine connects to a listening unix domain path. The ***path*** argument is a pointer to a character buffer containing the unix domain path of the listen host. Upon successful connection the routine will return a socket descriptor, which can be used as an argument to **`read`** and **`write`** to exchange information with the connected host program. Upon failure -1 will be returned.

Table 2: C Connection Library Routines

`int bs_sock_is_tcp(sd)`
`int sd;`
A connected socket descriptor (one returned by the **bs_accept** routine) may be tested to check if it is part of a TCP connection. The *sd* argument is the connected socket descriptor. If the connected socket descriptor is part of a TCP connection, then 1 is returned, otherwise 0 is returned.

`char *bs_get_tcp_name(sd, format)`
`int sd, format;`
It may be desired to obtain the hostname (or host address) of a connected socket descriptor (one returned by the **bs_accept** routine). This may be useful as a very basic security measure to prevent connections from unauthorized hosts. The *sd* argument is the connected socket descriptor. The *format* argument is either **BS_HOSTNAME**, or **BS_DOTTED**. The routine returns a pointer to a character buffer which either contains the hostname of the connected client or the IP address (in dotted decimal form) of the connected client, depending on the value of the *format* argument. This character buffer is static and will be overwritten on the next successive call to this routine.

`read(), write(), close();` The **read** and **write** and **close** routines are system calls provided by the operating system for reading from, writing to, and closing file descriptors and socket descriptors. Programs can write bytes to a socket descriptor, the connected program can then read those bytes from it's associated socket descriptor. Reading from a socket descriptor is slightly different than reading from a file, in that if the program attempts to read bytes from a descriptor for which no bytes are available, the program will block until these bytes are available. See any Unix system programming text for a treatment of **read** and **write**. All socket descriptors should be closed using the **close** system call when they are no longer needed and before the program exits. Note, that after closing a listening unix domain socket descriptor, the unique file it created at the specified path will still exist. The **unlink** system call should be used to remove it from the file system.

Table 2: C Connection Library Routines

3.4.1 Exception Handling for the C Routines

Many of the routines of the C connection library routines return -1 upon error. There may be one or many reasons why the routine failed. For example, when using **bs_tcp_connect** to connect to a host, the supplied hostname might not be spelled correctly. In this case the remote hostname will not be able to be resolved. Should an error occur, the program may check the value of the external integer variable **bs_errno**. The following table lists the possible values of the **bs_errno** variable along with the corresponding description of the error.

bs_errno	description
0	Can't resolve remote host name.
1	Can't create socket.
2	Can't connect to the server.
3	Can't resolve local host name.
4	Local hostname error.
5	Can't create a TCP socket.
6	Can't bind local address or path.
7	Listen failed.
8	Accept failed.

TABLE 3. Connection Error Codes

3.5 Tcl Connection Library Routines

In all of the libraries, wherever possible, there is a direct one-to-one mapping between each available C library routine and its Tcl library routine counterpart. It is intended that the programmer interface to both the C routine and the Tcl routine will as parallel as possible. The following is a list of the Tcl connection library routines along with a description of each routine. Before using these routines the file containing the library code must be sourced.

bs_tcp_listen *port* This routine listens to a TCP port on the local machine for connections. The ***port*** argument should be a positive integer where 1-1023 are reserved for superuser processes. Upon success the routine will return a socket descriptor. This socket descriptor is a handle to the listening port and will be used as an argument to other routines. Upon failure the routine raises a Tcl error.
bs_unix_listen *path* This routine listens to a UNIX path on the local machine for connections. The ***path*** argument is should be a string containing the path to a local file. This file should not exist on the system before this call is issued, as the call will cause the operating system to create a special zero sized socket file on the local system. The program which issues this routine as well as programs wishing to connect to it should all have read and write access to the directory where this special file exists. The routine will return a socket descriptor, upon success and will raise a Tcl error upon failure.

Table 4: Tcl Connection Library Routines

bs_accept *sd* This routine accepts a connection on a listening socket descriptor. The ***sd*** argument is the listening socket descriptor which was obtained by a call to **bs_tcp_listen** or **bs_unix_listen**. This routine will block if the connecting client has not yet attempted to connect, otherwise it will return immediately. The routine returns a new socket descriptor which is handle to the connected client, and can be used as an argument to **read** and **puts** to exchange information with the client. It is possible to accept multiple connections on the same listening socket descriptor, by calling **bs_accept** multiple times with the same argument. For each connecting client, **bs_accept** will return a new socket descriptor which is connected only with one client. The **bs_accept** routine will return -1 upon failure.
bs_tcp_connect *host port* This routine connects to a listening TCP port on a host. The ***host*** argument is a string containing the IP address of the host either as a hostname (i.e. "garcon.mit.edu") or in dotted decimal notation (i.e. "18.62.0.242"). Upon successful connection the routine will return a socket descriptor, which may be used as an argument to **read** and **puts** to exchange information with the connected host program. Upon failure the routine will raise a Tcl error.
bs_unix_connect *path* This routine connects to a listening unix domain path. The ***path*** argument is a string containing the unix domain path of the listening host. Upon successful connection the routine will return a socket descriptor, which can be used as an argument to **read** and **puts** to exchange information with the connected host program. Upon failure, a Tcl error will be raised.
bs_sock_is_tcp *sd* A connected socket descriptor (one returned by the **bs_accept** routine) may be tested to check if it is part of a TCP connection. The ***sd*** argument is the connected socket descriptor. If the connected socket descriptor is part of a TCP connection, then 1 is returned, otherwise 0 is returned.
bs_get_tcp_name *sd* This routine returns the IP address in dotted decimal notation of a connected socket descriptor (on returned by the **bs_accept** routine). The ***sd*** argument is the connected socket descriptor.
`read, puts, close` The **read** and **puts** and **close** routines provided by Tcl for reading from, writing to, and closing file descriptors and socket descriptors. Programs may write strings to a socket descriptor, the connected program can then read those bytes from it's associated socket descriptor. All socket descriptors should be closed using the **close** routine when they are no longer needed and before the program exits. Note, that after closing a listening unix domain socket descriptor, the unique file it created at the specified path will still exist. The **exec rm** Tcl command can be used to remove it from the file system.

Table 4: Tcl Connection Library Routines

3.5.1 Exception Handling for the Tcl Routines

Whenever one of the Tcl library routines fails, a Tcl error is raised. This error can be trapped by the Tcl program by using the **catch** command. Once caught, the error message may be displayed to determine the cause of the error. This is the normal exception

handling technique used by Tcl.

3.6 Other Socket Types

While the connection support library provides support only for TCP sockets and Unix domain sockets, all of the higher level libraries of the CHAT suit will work with any type of stream-oriented socket. There are several Unix system calls which can be used for creating and establishing connections which use other types of sockets.

3.7 Connection Library Example: The Echo Server and Clients

In this section a small example will be presented. An line oriented echo server will be created which will run continuously on a host. The job of the server is to listen for connecting clients and accept connections. The server will read a complete line of text (a string followed by a newline character) from the connected client and then write the line back to the client. The server will continue reading lines from the client until the client exits. At that point, the server will close it's connection and wait for another client to connect. The server runs continuously, until a client sends it a line of text containing the string "CLOSE_SERVER". The server will then close it's connection and exit.

In the example, the client will also be presented. The job of the client is to connect to the echo server. It then will read a complete line from *stdin* and write it to the echo server. It will then read the reply line from the echo server and print it out to *stdout*. This continues until there is an *eof* on *stdin*, at which point the client closes its connection and exits.

3.7.1 The Server Code

The example server code is written in C and uses the C connection library routines. The code is shown in table 5. The program includes "basic_sockets.h" which is the header file

```
/* The following is the code for a line based echo server */

#include <stdio.h>
#include <string.h>

/* The following includes the header file for the connection library */
#include "basic_sockets.h"

#define PORT 9656
#define BUFFER_SIZE 512
```

Table 5: Echo Server C Code

```
extern int bs_errno; /* global error code */

main() {
  int listening_socket, accepted_socket, length;
  char c, line[BUFFER_SIZE];

  /* attempt to set up listening socket */
  if ((listening_socket = bs_tcp_listen(PORT)) < 0) {
    fprintf(stderr, "Error code: %d\n", bs_errno);
    exit(1);
  }

  /* This is an infinite loop that will allow client after client to
     connect */

  for (;;) {

    /* attempt to accept connection from client */
    if ((accepted_socket = bs_accept(listening_socket)) < 0) {
      fprintf(stderr, "Error code: %d\n", bs_errno);
      close(listening_socket);
      exit(1);
    }

    /* continually read character from client until a complete line is
       received or the client dies */
    length = 0;
    while (read(accepted_socket, &c, 1) == 1) {
      /* got a character */
      line[length++] = c;
      if (c == '\n') { /* complete line */

        write(accepted_socket, line, length); /* echo line to client */

        /* check line for CLOSE_SERVER message */
        if (strncmp(line, "CLOSE_SERVER\n", length) == 0) {
          close (listening_socket);        /* close sockets and exit */
          close (accepted_socket);
          exit(0);
        }

        length = 0;
      }
    }

    /* read failed, client has died, close connection and repeat the loop */
    close(accepted_socket);
  }
}
```

Table 5: Echo Server C Code

for the connection library routines. The program starts by obtaining a listening TCP socket on port 9656, which was arbitrarily chosen. Then it accepts a connection on that listening socket. Once the program has successfully done this, it starts reading characters, one at a

time from the connected client and saves them in a character buffer. Although, the character buffer is of a fixed size, for simplicity of the example, the program does no checking to see whether the buffer is full. When a newline character is received by the server, it then echoes the received line back to the client, by writing to the connected socket descriptor. If the line received contains a "CLOSE_SERVER" message, then the server will close its sockets and exit. Otherwise, the server will simply read lines from the connected client and echo them back. If when reading a character, the **read** routine returns 0 (causing the while loop to terminate), then this indicates that the client program has died. When this occurs, the server closes the connection from that client and accepts a connection from another client. Note that a limitation of this server is that it can only service one connected client at a time.

3.7.2 The Client Code

Two echo client programs will be presented in this section. For simplicity, both programs will assume that the server is running on the host "garcon.mit.edu" and listening to port 9656. The first client is written in C and shown below in table 6. The client starts by

```
* The following is the code for the line-based echo client */

#include <stdio.h>
#include <string.h>
#include "basic_sockets.h"

#define SERV_PORT 9656
#define SERV_NAME "garcon.mit.edu"

#define BUFFER_SIZE 512

extern int bs_errno; /* global error code */

main() {
  int connected_socket, length, status;
  char c, line[BUFFER_SIZE];

  /* attempt to connect to the server */
  if ((connected_socket = bs_tcp_connect(SERV_NAME, SERV_PORT)) < 0) {
    fprintf(stderr, "Error code: %d\n", bs_errno);
    exit(1);
  }

  /* get a line from stdin */
  while(fgets(line, BUFFER_SIZE, stdin) != NULL) {
    length = strlen(line);
    write(connected_socket, line, length); /* write line to server */
```

Table 6: Echo Client C Code

```
    /* read line from server */
    length = 0;
    while ((status = read(connected_socket, &c, 1)) == 1) {
      line[length++] = c;
      if (c == '\n') { /* complete line */
        fprintf(stdout, "%s", line); /* print line to stdout */
        if (strncmp(line, "CLOSE_SERVER\n", length) == 0) {
          close(connected_socket);
          exit(0);
        }
        length = 0;
        break;
      }
    }
    if (status == 0) { /* server died */
      fprintf(stderr, "Server Died\n");
      close(connected_socket);
      exit(1);
    }
  }
  /* eof on stdin */
  close(connected_socket);
  exit(0);
}
```

Table 6: Echo Client C Code

attempting to connect to the server. If that succeeds, then the client will get a line from *stdin* and write it to the server. The server then echoes the line back to the client which subsequently prints it to *stdout*. The client will test the echoed line to see if it contains the "CLOSE_SERVER" message. If it does then the client will exit, since the server will no longer be running. Otherwise, the client will continue and the get line, write line, read echoed line, print line process will continue until there is an *eof* condition on *stdin*.

This echo client has been written in Tcl/TK and the code is shown below in table 7.

```
# The following is the code for the line based echo client

source "basic_sockets.tcl"

set SERV_PORT 9656
set SERV_NAME "garcon.mit.edu"

# Attempt to connect to server

if {[catch {set connected_socket [bs_tcp_connect $SERV_NAME $SERV_PORT]} err]
\
        != 0} {
    puts stderr $err
    exit
}

# Loop forever
```

Table 7: Echo Client Tcl Code

```
while {1} {

    set status [gets stdin line]; # Get a line from stdin

    # Check for eof
    if {$status == -1} {
        close $connected_socket
        exit
    }

    puts $connected_socket $line; # Write line to server
    set line ""; # Read line from server
    while {1} {
        set c [read $connected_socket 1]

        # Check to see if server died
        if {$c == ""} {
            close $connected_socket
            exit
        }

        # Check for complete line
        if {$c == "\n"} {
            puts stdout $line
            # Check if line contains "CLOSE_SERVER"
            if {$line == "CLOSE_SERVER"} {
                close $connected_socket
                exit
            }
            break
        }

    set line $line$c
    }
}
```

Table 7: Echo Client Tcl Code

In order for the program to run, it must source the "basic_sockets.tcl" file, which contains the source code for the Tcl connection library routines. The Tcl client is very similar to the C client, however, the exception handling mechanism is slightly different. The Tcl program uses the Tcl **`catch`** command to trap for an error when trying to connect to the server.

3.8 Summary

The connection support library, presented in this chapter, provides easy to use routines for establishing connections between two programs. A program may be allowed to have multiple open connections between it and other programs, however, for each connection to be successfully established there is a defined flow of events which must take place. The connection support routines, as they stand, provide no mechanism for determining whether

there exists new clients wishing to connect. Programs must simply execute the accept routine on a listening socket, and if there is a connecting client then the accept routine will return, otherwise it will block. The communications handling library, presented in the next chapter, will provide both a polling mechanism and an event-based mechanism for testing listening sockets for newly connecting clients. This will be vital in writing programs which allow an arbitrary number of clients to connect, such as the viewgraph server program, which was outlined in chapter 2.

The echo server and client programs are small programs which demonstrate a method of exchanging information between programs. These programs exchange lines of text, and the event leading up to a line being sent by a client, and the action after a line has been received by the server is very simple. In many cases the information to be exchanged amongst programs, and the subsequent actions will be much more complicated. The message packaging library will address this issue in chapter 5.

Chapter 4

The Communications Handling Library

The communications handling library is at the core of the CHAT suite. The library provides a mechanism for determining when new clients have connected. There are two approaches for doing this. One allows the program, at any time, to simply test whether a new client has connected. This is a polling mechanism. The other allows the program to bind it's own internal handling routine to a new client connection event. Whenever a new client connects, the program will be interrupted and its handling routine will automatically be called. This is an event-based mechanism.

Similar to determining whether new clients have connected, the connection handling library provides both a polling and event-based mechanisms for determining when new messages have arrived. In the polling situation, messages are transparently received and placed in a message queue. At any time the program may test the status of the message queue, and pop messages from the queue as desired. In the event-based situation, incoming messages are bound to an internal handling routine. Whenever a message arrives, the program is automatically interrupted and the internal handler routine is called. The library also provides a combination of the two approaches, by allowing the program to toggle between the polling mechanism and the event-based mechanism.

The routines provided by the communications handling library are flexible. They will work for a wide variety of message formats. It will be necessary to define what a mes-

sage is, and how to configure the routines to work with some standard message formats, as well as any new formats created by the programmer. The following figure shows what is provided to a messaging application by the communications handling library.

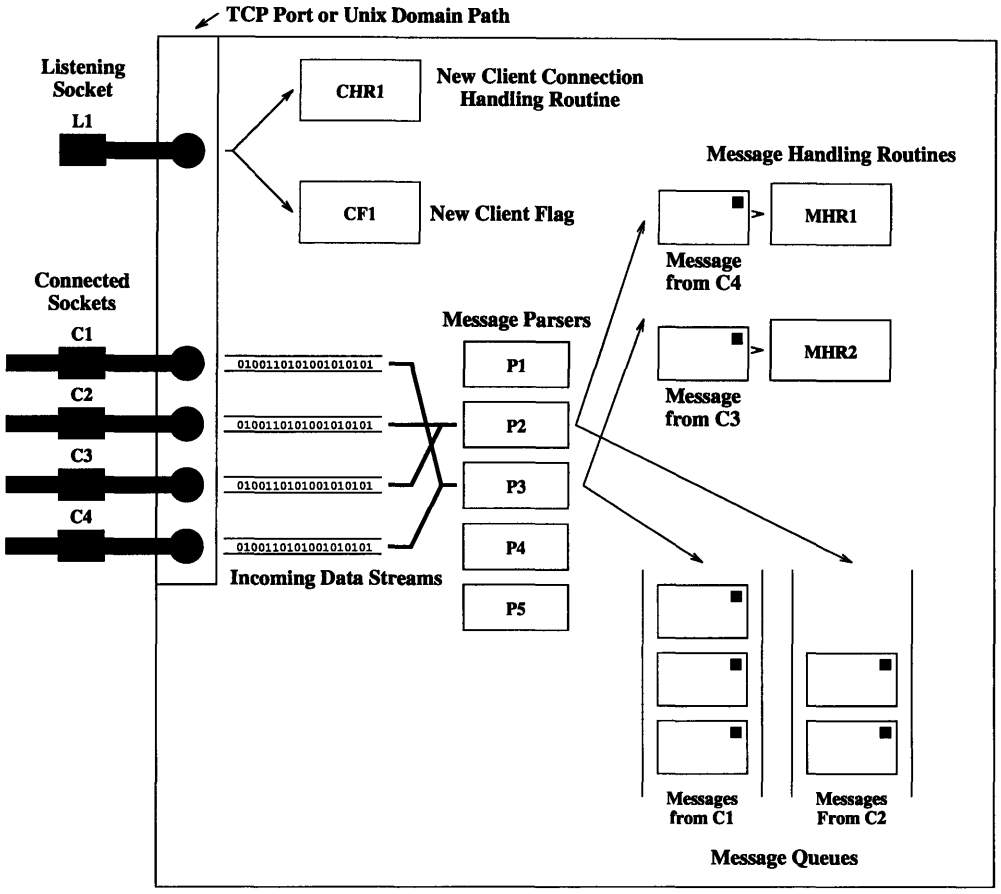


Figure 11: Communications Handling Library

4.1 Handling Newly Connected Clients

As a program runs, clients may attempt to asynchronously connect to it. If the program has no prior knowledge about how many simultaneous connections it will be handling, then the program must have some other mechanism for handling newly connected clients than what is provided by the connection support library.

4.1.1 Polling-Based Handling of Newly Connected Clients

A simple way to handle newly connecting clients is to allow the program to poll a listening socket for a connection. This allows a program to create a polling loop where the pro-

gram can on each iteration of the loop, check the status of the socket for new connections and then sleep for a fixed amount of time. If the program detects that there is a newly connected client, then it can accept a connection on the listening socket, and be guaranteed that the accept routine will not block. The **com_connected** roiutine is used to test the status of a listening socket for newly connected clients.

4.1.2 Event-Based Handling of Newly Connected Clients

A more sophisticated approach to handle newly connected clients, is for the program to register its interest in a listening socket. Associated with that interest is a subroutine. Whenever an new client connects to the program, that event will cause the program to be automatically interrupted, and the subroutine will be called. The subroutine can then accept a connection on the listening socket, and be guaranteed that the accept routine won't block. This event-based approach to handling new client connections is accomplished by using the **com_register_listen** routine.

4.1.3 C Library Routines for Handling Newly Connected Client

The C routines provided by the communications handling library for handling newly connected clients are specified below in table 8. The **com_register_listen** routine allows the program to associate one of its own subroutines, with the new client connection event. This handler subroutine must conform to a predefined prototype. Passed to the handler will be the actual socket descriptor of the listening socket for which a new client has connected. This allows the same handler routine to be used to handle connections on multiple listening sockets. For example, a program might have a listening Unix domain socket as well as a listening TCP socket. Both of these listening sockets may be registered with the same handler subroutine, such that a new client connection on either socket will cause the same subroutine to be called. Since the subroutine has the actual socket descriptor passed in as an argument, it can then simply accept the connection on that descriptor. The

handler routine must also return a void.

`int com_connected(sd)`
`int sd;`
Tests the listening socket descriptor ***sd*** to determine whether a new client has connected to that socket. Returns 1 if a new client has connected and 0 otherwise. If **`com_connected`** returns 1, then a call to **`bs_accept`** is guaranteed to accept a connection immediately without blocking.

`int com_register_listen(sd, handler)`
`int sd;`
`void (*handler)();`
This routine associates a handler routine which is to be called whenever a new client connects to the listening socket descriptor specified by the ***sd*** argument. The ***handler*** argument is a pointer to a function which returns a void. A pointer to a function in C is simply the name of the function. When this handler function is called, it will be passed the socket descriptor on which the new client has connected. It is guaranteed that a call to **`bs_accept`** on that socket descriptor within the handler routine will not block. The **`com_register_listen`** routine will return -1 on error, and 0 otherwise.

`void com_unregister(sd)`
`int sd;`
This routine, stops the handling of newly connected clients on the listening socket descriptor ***sd***.

Table 8: C Library Routines for Handling Newly Connected Clients

In table 9 is the skeleton code for a printer server program. The program simultaneously listens to a TCP socket and a Unix domain socket. Whenever a new client connects to either socket, the server reads data from the client and sends it to the printer. For simplicity, the skeleton program does no exception handling. The program starts by creating two listening sockets, and then registering each socket with the handling routine (in this example called *new_client*). After the sockets are register, the program enters a *while* loop, where it pauses, and blocks until an occur. When a new client event occurs, the program breaks out of the *pause*, and the handler routine is called. When the handler routine is completed, the program re-enters the *while* loop. All events which are handled by the routines in the communications handling library are guaranteed to not be preempted by other events. For example, during the time when the *new_client* handler routine is running it is possible for another new client connection to occur. When this happens, the *new_client* handler routine will finish handling the first event, and when it returns it will be immedi-

ately called with the next event.

```
/* Skeleton printer server program */
#include "basic_sockets.h"
#include "com_handler.h"

/* This routine is called whenever a new client connects to either the
 * listening TCP socket, or the listening Unix domain socket */
void new_client(sd)
int sd;
{
   int accepted_socket;
   accepted_socket = bs_accept(sd);    /* accept connection, this will not block */
   . . . . . .    /* read data from accepted_socket and send it to the printer */
   close(accepted_socket);          /* finished printing, close accepted socket */
}

main() {
   int listening_socket1, listening_socket2;
   listening_socket1 = bs_tcp_listen("garcon.mit.edu", 2454); /* Create TCP sock */
   listening_socket2 = bs_unix_listen("/tmp/pservsock");     /* Create Unix sock */
   com_register_listen(listening_socket1, new_client);      /* Register TCP sock */
   com_register_listen(listening_socket2, new_client);     /* Register Unix sock */
   while(1) pause();           /* Block until an event occurs, then repeat loop */
}
```

Table 9: C Skeleton Printer Server Program

4.1.4 Tcl Routines for Handling Newly Connected Clients

The Tcl routines for handling newly connected clients are parallel to the C routines presented earlier. The prototypes and descriptions of the Tcl routines are included below in table 10.

com_connected *sd* Tests the listening socket descriptor ***sd*** to determine whether a new client ha connected to that socket. Returns 1 if a new client has connected and 0 otherwise. If **com_connected** returns 1, then a call to **bs_accept** is guaranteed to accept a connection immediately without blocking.
com_register_listen *sd handler* This routine associates a handler routine which is to be called whenever a new client connects to the listening socket descriptor specified by the ***sd*** argument. The ***handler*** argument is name of the associated handling function. When this handler function is called, it will be passed the socket descriptor on which the new client has connected. It is guaranteed that a call to **bs_accept** on that socket descriptor within the handler routine will not block. The **com_register_listen** routine has no return value.
com_unregister *sd* This routine, stops the handling of newly connected clients on the listening socket descriptor ***sd***.

Table 10: Tcl Library Routines for Handling Newly Connected Clients

As in the C case, a handler routine which is specified in the com_register_listen routine must conform to a predefined prototype. The Tcl handler routine should expect one argument which will be the socket descriptor of the listening socket on which a new client has connected. The handler routine should return no value. The following is the skeleton version of the print server program. For simplicity of the example, no exception handling code is included. The only noticeable difference between the C version and the Tcl version is that the Tcl program does not use the *while(1) pause()* loop construct, which causes the program to block until an event occurs. Instead the internal event loop provided by TK (Tcl's graphical toolkit) automatically provides this functionality.

```
# Skeleton printer server program
source "basic_sockets.tcl"
source "com_handler.tcl"

# This routine is called whenever a new client connects to either the
# listening TCP socket, or the listening Unix domain socket */
proc new_client {sd} {
   set accepted_socket [bs_accept $sd]; # accept Connection, this will not block
   . . . . . ; # read data from accepted_socket and send it to the printer
   close $accepted_socket; # finish printing, close accepted socket
}

set listening_socket1 [bs_tcp_listen "garcon.mit.edu" 2454]; # Create TCP sock
set listening_socket2 [bs_unix_listen("/tmp/pservsock")];    # Create Unix sock
com_register_listen $listening_socket1 new_client;           # Register TCP sock
com_register_listen $listening_socket2 new_client;           # Register Unix sock
```

Table 11: Tcl Skeleton Printer Server Program

4.2 Definition of a Message

Data communicated over socket connections must be containable within a serial buffer of bytes. A message is a notification or request of some kind including data and information which is to be communicated among connected programs. Since these messages will be communicated over socket connections, ultimately they must be containable within a serial buffer of bytes. There is another constraint on the format of a message. Since socket communication is stream oriented several buffers of characters which are written in succession to the stream will appear as a single concatenation of messages to the receiving program. The receiving program must be able to detect where one message ends in the stream and the next message begins. To accomplish this, messages must formatted such that they may be decoded and extracted from the stream. For example, consider a message format containing ASCII text followed by a newline character. The newline character acts

as a separator between successive messages, and when the receiving program sees a newline character, it knows that a complete message has arrived. The message handling routines provided in the communications handling library will work on a wide variety of message formats. When using these handling routines it will be necessary to specify the message format. This is done by specifying an appropriate parser for the message format. The parser, when given a buffer containing possibly many concatenated messages, can parse a single message from the buffer. The library comes with several standard parsers, however, a parser interface will be defined so that new message parsers may be added as required. Note, the parsers which are being described here are only used to parse a complete message from a stream of messages and are not being used to parse out individual pieces of information contained within the messages.

4.3 Standard Message Format and Parsers

In table 12 is a list of the standard message formats and parsers which are available with the communications handling library. It is intended that these formats will provide, in most cases, enough of a basis for a wide range of message passing applications to be built. A programmer simply must choose a suitable message format for the application from those which are readily available.

Format	Parser	Description
line	`ch_parse_line`	Considers a message to be a string of ASCII characters followed by a newline character. The parser parses the string and truncates the newline character.
fline	`ch_parse_fline`	Same as the line format, however, the parser parses the string and includes the newline character.
brace	`ch_parse_brace`	Considers a message to be ASCII strings of characters contained within matched sets of curly braces, arbitrarily nested. The backslash character is considered an escape character. A curly brace preceded by a backslash is not considered a matching brace. A backslash character followed by a backslash character is simply considered to be a single backslash. The parser parses the message, excluding the matching braces comprised of the first curly brace and the last curly brace. Example, "{this {is} {a {message}}}" will get parsed as "this {is} {a {message}}".

Table 12: Standard Message Formats

Format	Parser	Description
fbrace	`ch_parse_fbrace`	Same as the brace format except the parser includes the matching braces comprised of the first curly brace and the last curly brace.
binary[a]	`ch_parse_binary`	This format is made of binary bytes of data, assembled together in a special way. This format will be discussed in more detail in chapter 5 and in an appendix.

Table 12: Standard Message Formats

a. This format is not available in the Tcl version of the communications handling library, since Tcl is not capable of handling binary data.

4.3.1 Creating a New Parser

Additional parsers may be written and used with the message handling routines. To create a new parser it must adhere to a standard prototype. The following is the prototype which C parsers should conform to:

```
int a_new_parser(buff, size, start, length, next)
char *buff;
int size, *start, *length, *next;
{
   /* your parser code goes here */
}
```

The parser is passed a pointer *buff*, which is a pointer to a character buffer containing all bytes which have been read from the stream. This buffer may contain an incomplete message, or several messages concatenated together. The *size* argument is the number of bytes which is contained in this character buffer. The job of the parser is to examine the buffer and determine if at least one complete message is contained within the buffer. The variables *start*, *length*, and *next* are all pointers to pre-allocated integers. If the buffer contains a complete message, then the parser should set **start* to be the offset within the buffer of the first character in the message. It should set **length* to be the number of characters in the message (possibly 0). It should set **next* to be the offset within the buffer of where a new buffer should be formed for the next successive call to the parser. The parser should then return 1 if it was successful in parsing a message from the buffer. Otherwise the parser should return 0. The value of **next* cannot be greater than the *size* argument. If it must be greater than the *size* argument, then the parser should return 0. The parser will get a chance to try again on the next call which will be on a greater sized buffer. The idea of

having three values, **start*, **length*, **next*, to specify a message is to allow for any type of decodable message format which may or may not have a header or trailer or both. If a message format contains a header or trailer or both, then by properly setting these values, the parser may (if it wishes) extract out the internal and meaningful section of the message and throw away the header and trailer.

Consider an example for a new parser. We will create a message format where the message is designed such that it will always begin with the two characters "<<" and the end of the message is denoted by the two characters ">>". We will allow arbitrary characters to occur between separate messages (possibly whitespace, etc.). For this format, the important information resides between the less-thans and greater-thans. This will be called the "llgg" parser. The following table shows some possible scenarios of inputs to the parser, and what the parser should do in response to those inputs.

Input *buff	Input size	Result of Parser
`"---<<reque"`	10	The parser determines that there is not a complete message in the buffer and returns -1.

`"---<<request value a>>"`	22	The parser determines that there is a complete message in the buffer and sets:"
		*start = 5;
		*length = 15;
		*next = 22;
		The parser then returns 1.

`"--<<ack>>--<<verify-"`	20	The parser recognizes a complete message and sets:
		*start = 4;
		*length = 3;
		*next = 9;
		The parer then returns 1. Note, in this case the parser could have set *next to be anything between and including 9 and 11;

`"-<<>>-"`	6	The parser recognizes a complete message and sets:
		*start = 3;
		*length = 0;
		*next = 5;
		The parser then returns 1.

Table 13: Example Parser Scenarios

The following is a possible C implementation of the parser. Note that the parser should

only be used to read the buffer which is passed to it, and should not modify it in any way.

```
/* This is an example parser */
int ch_parse_llgg(buff, size, start, length, next)
char *buff;
int size, *start, *length, *next;
{
   int i;
   for (i=1; i<size; i++) {
      if ((buff[i] == '<') && (buff[i-1] == '<'))   /* found "<<" */
         *start = i + 1;
      if ((buff[i] == '>') && (buff[i-1] == '>')) { /* found ">>" */
         *length = (i - 1) - *start;
         *next = i + 1;
         *return 1;
      }
   }
   return 0;
}
```

Table 14: Example Parser Implementation in C

Creating a Tcl parser is very similar, to creating a C parser. Passed to the Tcl parser routine is a string containing the buffer, and the size of the buffer. The Tcl routine returns an empty string to indicate that it could not successfully parse a message from the buffer. Otherwise it returns a Tcl list containing the three values for *start*, *length*, and *next*. The following is an example implementation of a Tcl version of the "llgg" parser.

```
# This is an example parser
proc ch_parse_llgg {buff size} {
   set tmp [string first "<<" $buff];  # Look for "<<"
   if {$tmp == -1} {
      return {}
   } else {
      set start [expr $tmp + 2]
   }

   set tmp [string first ">>" $buff];  # Look for ">>"
   if {$tmp == -1} {
      return {}
   } else {
      set length [expr $tmp-1 - $start]
      set next [expr $tmp+2]
      return [list $start $length $next]
   }
}
```

Table 15: Example Parser Implementation in Tcl

4.4 Handling of Incoming Messages

Messages are comprised of bytes of data which can be received on a socket and buffered by the operating system. A program can read bytes from the buffer, determine if there are

complete messages available, and then act on those messages. The message handling routines, provide a cleaner and transparent mechanism for handling incoming messages to a program. The program registers its interest in incoming messages on a particular connected socket descriptor. While registering interest, the program must also specify the format of the incoming messages on that descriptor. This is done by simply choosing (or creating) the appropriate message parser for the desired message format. Once a connected socket is registered, bytes received on it will automatically be read by the program as they become available and attempted to be parsed by the message parser. If a complete message can successfully be parsed, the user's handler routine may be called to act on the message, or the message may automatically placed in a message queue. The action that occurs when a message is received depends on how the program has configured its interest. The message handling library will also be able to automatically detect the death of a connected socket, and will allow the program to act appropriately when this occurs.

4.5 Event-Based Message Handling

The event-based messaging approach allows a program to bind messaging events on a connected socket descriptor to a handling routine. There, are two types of events which may occur. The first is a message arrival. Whenever a new message arrives, the program will be interrupted and the user's routine bound to that event will automatically be called. The other type of event which may occur is the death of a connected socket descriptor. Whenever this event occurs, the program will also be interrupted and the user's routine which is bound to that event will be called. This routine may then unregister the dead descriptor, and close it, as well as perform any other clean-up which should be done when a connected client program dies (possibly unexpectedly). Many client-server programs may be completely event-based. They remain in an idle for the great majority of the time, and only exit the idle state to handle evenst. They then return to the idle state after the event has been handled. The event-based message handling routines provided make this kind of event-based program easy to write. While a handler is being executed, it is possible that another event will occur during that time. When this occurs, the first handler will complete, and then another handler will be called to handle the most recent event. The message arrived event, and connection died events are level-triggered.

4.5.1 C Library Routines for Event-Based Message Handling

There are two routines which provide support for event-based messaging. The **com_register_event** routine is used to bind messaging events on a connected descriptor to handler routines. The **com_unregister** routine is used to unbind these events from being handled. In table 16 the event-based messaging routines are described in more detail.

Handler routines are written by the programmer and must conform to a standard prototype. The following is an example handler routine which handles a newly arrived message.

```
int a_new_handler(message, size, sd)
char *message;
int size, sd;
{
   /* your message handling code goes here */
}
```

The ***message*** argument is a pointer to a character buffer containing the newly arrived message. The handling routine may read bytes from this character buffer. The ***size*** argument is the number of bytes contained in the message buffer. Note, the byte just following the message buffer is guaranteed to be an ASCII null (value 0). Although this byte is not part of the message itself, it may be helpful when processing ASCII based messages. The message buffer will automatically be destroyed when the handler returns. If it is desired for a message to persist after the handler returns, then it should be copied by the handler routine. The ***sd*** argument is the connected socket on which the message was received. This allows for the possibility of multiple connected sockets to be bound to the same handling routine. The following is an example handler routine which handles a connected socket death.

```
int a_new_death_handler(sd);
int sd;
{
   /* your socket death handling code goes here */
   com_unregister(sd); close(sd);
}
```

The ***sd*** argument is the connected socket descriptor which has died. In addition to the application specific clean-up which is performed within the handler, the handling routine

should also **com_unregister** the connected socket and **close** it.

`int com_register_event(sd, parser, handler, death_handler)`
`int sd, (*parser)();`
`void (*handler)(), (*death_handler)();`
This routine associates a handler routine which is to be called whenever a new message arrives on the connected socket descriptor specified by the ***sd*** argument. The routine also associates a handler routine which should be called if the program on the other end of the connected socket dies. The ***parser*** argument is a pointer to a message parsing routine and specifies the message format of the incoming messages over the connected socket. The ***handler*** argument is a pointer to the associated message handling routine which will be called whenever a new message arrives on the connected socket. When this handler routine is called, it will be passed, as arguments, a pointer to a character buffer containing the newly arrived message, the size of the message, and the socket descriptor over which the message was received. The ***death_handler*** argument is a pointer to the associated handling routine which will be called if the program on the other end of the connected socket dies. When this routine is called it will be passed as an argument the connected socket descriptor which has died. Recall that in C, pointers to routines (or functions) are simply the name of the routine. The **com_register_event** routine will return -1 on error, and 0 otherwise.

`void com_unregister(sd)` This routine stops the handling of messaging events on socket descriptor ***sd***.

Table 16: C Library Routines Event-Based Message Handling

4.5.2 Tcl Library Routines for Event-Based Message Handling

The Tcl library routines for Event-Based messaging are exactly analogous to the C library routines. These are described in table 17. The following is a Tcl handler routine to handle a newly arrived message.

```
proc a_new_handler {message size sd} {
{
   # your message handling code goes here
}
```

In the Tcl case, the ***message*** argument is a string which contains the newly arrived message. The ***size*** and ***sd*** arguments have the same meaning as in the C case. The following is and Tcl handler routine to handler a connected socket death.

```
proc a_new_death_handler {sd} {
{
   # your socket death handling code goes here
   com_unregister $sd; close $sd
}
```

The handler should **com_unregister** and **close** the dead socket descriptor.

com_register_event *sd parser handler death_handler* This routine associates a handler routine which is to be called whenever a new message arrives on the connected socket descriptor specified by the ***sd*** argument. The routine also associates a handler routine which should be called if the program on the other end of the connected socket dies. The ***parser*** argument is the name of a message parsing routine and specifies the message format of the incoming messages over the connected socket. The ***handler*** argument is the name of the associated message handling routine which will be called whenever a new message arrives on the connected socket. When this handler routine is called, it will be passed, as arguments, a string containing the newly arrived message, the size of the message, and the socket descriptor over which the message was received. The ***death_handler*** argument is the name of the associated handling routine which will be called if the program on the other end of the connected socket dies. When this routine is called it will be passed as an argument the connected socket descriptor which has died. The **com_register_event** routine has no return value
com_unregister *sd* This routine stops the handling of messaging events on socket descriptor ***sd***.

Table 17: C Library Routines for Event-Based Message Handling

4.5.3 An Event-Based Messaging Example

In chapter 3, the line-based echo server was presented and implemented. The implemented server could only handle one connected client at any given time. Here, we will extend the echo server so that it can handle arbitrary simultaneous clients. We will do this in an event based manner. The server will start-up and create a listening socket. It will then associate a *new_client* handler, which will be called whenever a new client wishes to connect. The program will remain in an idle state waiting for client connections. When new clients connect, the *new_client* handler will accept the connection from the client, and register the newly connected socket to handle messaging events. The *fline* message format will be specified. Whenever a connected client sends a message, the *new_message* handler will echo it back to the client and test it to determine if it is a "CLOSE_SERVER message". If it is the server will exit, otherwise it will return to the idle state. The server will handle client deaths with the *dead_client* handler which simply unregisters and closes the associated socket descriptor.

Although, this echo server can handle multiple clients simultaneously, the code is arguably as simple as the single client handling server which was presented in chapter 3. the following is the C code for the example line-based multiuser echo server. Exception

handling is omitted from the code for simplicity.

```
/* The following is the code for a multiuser line based echo server */

#include <stdio.h>
#include <string.h>

/* The following includes the header file for the connection library, and
the communication handling library. */
#include "basic_sockets.h"
#include "com_handler.h"

#define PORT 9656
#define BUFFER_SIZE 512

/* The following three routines are the event handlers */

void new_message(message, size, sd)  /* This handles new incoming messages */
char *message;
int size, sd;
{
  write(sd, message, size); /* echo message back to client */

  /* exit server if it is the "CLOSE_SERVER" message */
  if (strcmp(message, "CLOSE_SERVER\n") == 0)
    exit(0);
}

void dead_client(sd) /* This handles a client death */
     int sd;
{
  com_unregister(sd); close(sd); /* Stop handling and close descriptor */
}

void new_client(sd)  /* This handles new clients connecting */
int sd;
{
  int accepted_socket;
  accepted_socket = bs_accept(sd);  /* accept connection from client */

  /* register message handling on the newly connected client */
  com_register_event(accepted_socket, ch_parse_fline,
                     new_message, dead_client);
}

main() {
  int listening_socket;

  listening_socket = bs_tcp_listen(PORT);  /* get listening socket */
  com_register_listen(listening_socket, new_client); /* register socket */
  while(1) pause(); /* Wait in an idle state for events to occur */
}
```

Table 18: Example of Event-based Multiuser Echo Server

4.6 Queue-Based Message Handling

In the queue-based approach, a program may register its interest in a connected socket

descriptor. All incoming messages on that descriptor will be transparently received by the program and placed in a message queue. At any time the program may check the status of the message queue and pop messages from the queue. As in the event-based case, the programmer chooses a message format for the incoming messages by specifying the appropriate message parser.

4.6.1 C Library Routines for Queue-Based Message Handling

A program calls the **com_register_queue** routine to allow messages to be automatically received on a connected socket descriptor and queued. As it desires the program my issue the **com_stat** call to determine the number of messages in the queue, or the **com_died** call to determine if the connected socket has died. If the program wishes to pop a message from the queue, it may issue the **com_pop** call. Finally, when the program no longer wishes for messages to automatically be queued, the **com_unregister** call is used to turn off message handling on the connected socket descriptor. The following table describes the C library interface to these routines.

`int com_register_queue(sd, parser)`
`int sd, (*parser)();`
This routine expresses interest in messages arriving on the connected socket descriptor specified by the ***sd*** argument. The ***parser*** argument is a pointer to a message parsing routine and specifies the message format of the incoming messages over the connected socket. Newly arrived messages will automatically be queued. The **com_register_queue** routine will return -1 on error, and 0 otherwise.

`int com_stat(sd)`
`int sd;`
This routine checks the status of the message queue associated with the connected socket descriptor ***sd***. If there are any messages in the message queue, the **com_stat** routine will return a positive integer indicating the number of messages in the queue. If there are no messages in the queue, then 0 will be returned when the connected socket descriptor is alive and -1 will be returned when the connected socket descriptor is dead.

`int com_died(sd)`
`int sd;`
This routine returns 1 if the connected socket descriptor ***sd*** is dead, and 0 otherwise.

Table 19: C Library Routines for Queue-Based Message Handling

`char *com_pop(sd, size)`
`int sd, *size;`
This routine pops the earliest received message from the message queue associated with connected socket descriptor ***sd***. A pointer to a character buffer is returned. The character buffer contains the message. If there are no messages in the queue and the socket descriptor is alive, then **`com_pop`** will block until a message arrives in the queue[a]. If there are no messages in the queue and the descriptor is dead, then NULL will be returned. If a non-null pointer to a pre-allocated integer passed in as the ***size*** argument, it will get filled by the routine with the size of the message in bytes. It is guaranteed that an ASCII null (value 0) will always follow the last byte of the message. The responsibility is left to the programmer to free the message buffer when it is no longer needed. This is accomplished by issuing a **`free`** call.

`void com_unregister(sd)` This routine stops message handling on socket descriptor ***sd***.

Table 19: C Library Routines for Queue-Based Message Handling

a. The current implementation of this routine, prohibits it from blocking within the event handler of another socket descriptor as this will cause a deadlock situation. As a result, within a handler of another socket descriptor, the program should test the queue with `com_stat` to make sure that messages are available in the queue before issuing the `com_pop` call.

4.6.2 Tcl Library Routines for Queue-Based Message handling

The Tcl version of the communications handling library provides an analogous set of routines for queue-based message handling. These are described in table 20.

`com_register_queue` *`sd parser`* This routine expresses interest in messages arriving on the connected socket descriptor specified by the ***sd*** argument. The ***parser*** argument is the name of a message parsing routine and specifies the message format of the incoming messages over the connected socket. Newly arrived messages will automatically be queued. The **`com_register_queue`** routine returns no value.
`com_stat` *`sd`* This routine checks the status of the message queue associated with the connected socket descriptor ***sd***. If there are any messages in the message queue, the **`com_stat`** routine will return a positive integer indicating the number of messages in the queue. If there are no messages in the queue, then 0 will be returned when the connected socket descriptor is alive and -1 will be returned when the connected socket descriptor is dead.
`com_died` *`sd`* This routine returns 1 if the connected socket descriptor ***sd*** is dead, and 0 otherwise.

Table 20: Tcl Library Routines for Queue-Based Message Handling

com_pop *sd ?size?* This routine pops the earliest received message from the message queue associated with connected socket descriptor ***sd***. A string is returned which contains the message. If there are no messages in the queue and the socket descriptor is alive, then **com_pop** will not return until a message arrives in the queue[a]. If there are no messages in the queue and the descriptor is dead, then an empty string will be returned. The ***size*** argument is optional and is the name of a variable. If it is passed as an argument then the value of that variable will be set with the size of the message in bytes.
com_unregister *sd* This routine stops message handling on socket descriptor ***sd***.

Table 20: Tcl Library Routines for Queue-Based Message Handling

a. While **com_pop** is waiting for a message to arrive, idletasks will be processed, thus avoiding a lock-up of the display

4.7 Combination Message Handling

Throughout a program it is entirely possible to have some socket descriptors use an event-based message handling approach while others use a queue-based handling approach. However, there may be a situation where it would be desirable to have both capabilities on a single socket descriptor with the ability to, at any time, toggle between the two approaches. The communication handling library provides routines which support this kind of combination message handling.

4.7.1 C Library Routines for Combination Message Handling

A connected socket descriptor is registered using the **com_register_notify** command. The message handling starts off in the queue-based mode and the **com_stat**, **com_died**, **com_pop** routines will be functional. The program can then issue the **com_notify_on** routine, to switch from queue-based message handling to event-based message handling. If there are any messages in the message queue at this point, the handler routine will be called successively on each message in the queue until the queue is empty. Then the message handler will be called as each new message arrives. The program can switch back to queue-based messaging by issuing the **com_notify_off** call. The following table contains a description of the C interface to the combination message

handling routines.

`int com_register_notify(sd, parser, handler, death_handler)`
`int sd, (*parser)();`
`void (*handler)(), (*death_handler)();`
This routine has exactly the same interface as the **com_register_event** routine, however, it registers a connected socket descriptor for both event-based and queue-based message handling. Once registered, the initial state of message handling will be queue-based. The **com_stat**, **com_died**, and **com_pop** routines will all be functional.

`void com_notify_on(sd)`
`int sd;`
This routine puts the connect socket descriptor, ***sd***, into an event-based message handling mode. In this mode, the registered ***handler*** and ***death_handler*** routines will be called as message events occur on the socket descriptor. If there are any messages in the message queue when the message handling mode is switched, then the queue will be emptied by calling the handler routine successively on each message until the queue.

`void com_notify_off(sd)`
`int sd;`
This routine puts the connected socket descriptor, ***sd***, into a queue-based message handling mode. In this mode, messages will be place in the message queue as the arrive. The **com_stat**, **com_died**, and **com_pop** routines can then be used to access the messages in the queue.

`void com_unregister(sd)` This routine stops messaging handling on socket descriptor ***sd***.

Table 21: C Library Routines for Combination Message Handling

4.7.2 Tcl Library Routines for Combination Message Handling

The Tcl version of the communication handling library provides a parallel set of combination message handling routines. These are described in table 22.

com_register_notify *sd parser handler death_handler* This routine has exactly the same interface as the **com_register_event** routine, however, it registers a connected socket descriptor for both event-based and queue-based message handling. Once registered, the initial state of message handling will be queue-based. The **com_stat**, **com_died**, and **com_pop** routines will all be functional.
com_notify_on *sd* This routine puts the connect socket descriptor, ***sd***, into an event-based message handling mode. In this mode, the registered ***handler*** and ***death_handler*** routines will be called as message events occur on the socket descriptor. If there are any messages in the message queue when the message handling mode is switched, then the queue will be emptied by calling the handler routine successively on each message until the queue is empty.

Table 22: Tcl Library Routines for Combination Message Handling

com_notify_off *sd* This routine puts the connected socket descriptor, ***sd***, into a queue-based message handling mode. In this mode, messages will be place in the message queue as the arrive. The **com_stat**, **com_died**, and **com_pop** routines can then be used to access the messages in the queue.
com_unregister *sd* This routine stops messaging handling on socket descriptor ***sd***.

Table 22: Tcl Library Routines for Combination Message Handling

4.8 Summary

This chapter has presented the communications handling library of the CHAT suite. The library provides many flexible ways of handling communications. First, it provides a polling and event-based approach to handling new client connections. Programs which require connections on multiple sockets, or multiple connections on a single socket can be greatly simplified with the routines provided. Second, the library provides support for messages of varying formats and is not constrained to use single message format. Some standard message formats are provided and wide variety of others may be added by designing an appropriate message parser for the particular format. Finally, connected socket descriptors may be registered for event-based or queue-based message handling as well as a combination of the two. These routines are at the core of the CHAT suite. By configuring message handling in the appropriate way, the routines are designed to be integrated into just about any existing or newly created framework.

Chapter 5

The Message Packaging Library

In some cases a very simple message format is sufficient for an application. This message format may be as simple as a line of ASCII text. In many cases, however, it is desired to package data of varying types and lengths within a single message. For example, if a message contains the data for an image, it may be necessary to package the image data, as well as other information, such as the height, width and name of the image. The message packaging library provides two message formats which allow entries of singles or vectors of various primitive types to be packaged together to form a complete message. Ultimately the message must be containable in a serialized buffer of bytes so that it may be transmitted over a network (or even saved to a file). Direct manipulation of a buffer of is difficult. The approach of the message packaging library is to provide a programmer's interface, whereby a message data object may be created and slots within the object may be set. This message data object is stored as an internal data structure, with several available methods for manipulating it. All manipulation is done on the message data object by using the using library routines. At any time the object may be converted to a formatted serial buffer of bytes. This conversion from a data structure to a serial buffer is known as data marshalling. Likewise, the formatted serial buffer of bytes may be converted back into a message data object for further manipulation. The emphasis of the message packaging library is on the interface for constructing messages, and the actual data structures and message for-

mats used by the library is of little importance to a user of the library.

5.1 The World of Message Data Objects

The message packaging library maintains a global world of message data objects. Objects may be created and removed from the world at any time. The number of objects which can be contained in the world is arbitrary and limited only by system memory. Each message object is referenced by its name. A message object name is a case sensitive ASCII string which must begin with a letter and contain only letters, numbers, and underscores. Object names are unique in a namespace global to the program. Each object can have an arbitrary number of slots. Slots are referenced by name, following the same convention as message object names. Slot names are unique to the individual object and need not be unique to all objects. A slot contains data, which is a single or vector of a primitive type. There are twelve supported types which are described in table 23. The supported types are based on those supported by TIFF [Ald92], a tag-based file format for storing and interchanging raster images.

Type	Name	Description
1	OB_BYTE	8-bit unsigned integer.
2	OB_ASCII	8-bit byte containing 7-bit ASCII code; the last byte must be NUL (0).
3	OB_SHORT	16-bit (2-byte) unsigned integer.
4	OB_LONG	32-bit (4-byte) unsigned integer.
5	OB_RATIONAL	Two OB_LONGs: the first represents the numerator of a fraction; the second, the denominator.
6	OB_SBYTE	8-bit signed (twos complement) integer.
7	OB_UNDEFINED	8-bit byte containing anything.
8	OB_SSHORT	16-bit (2-byte) signed (twos complement) integer.
9	OB_SLONG	32-bit (4-byte) signed (twos complement) integer.
10	OB_SRATIONAL	Two OB_SLONGS: the first represents the numerator of a fraction; the second, the denominator.
11	OB_FLOAT	Single precision (4-byte) floating point number.
12	OB_DOUBLE	Double precision (8-byte) floating point number.

Table 23: Data Types

5.2 C Library Routines for Message Packaging

The C library routines for packaging messages are rich. The **object_create**, **object_rm**, and **object_mv** routines are used to create, remove and rename message data objects. The **object_slot_set** routine is used to package a new slot of data into the message, and similarly, the **object_slot_get** routine is used to query a slot of data from a message. The two routines **ascbuf_from_object** and **binbuf_from_object** are used to convert a message data object into a serial buffer of bytes. Both a binary buffer format and an ASCII buffer format can be produced[1]. Appropriate parsers exist for the two formats and they can be used in conjunction with the communications handling library. When using the binary buffer format, the **ch_parse_binary** parser should be used and when using the ASCII buffer format the **ch_parse_fbrace** parser should be used. When one of these formatted buffers is recieved by a program, the program may call the **object_from_binbuf** or **object_from_ascbuf** to convert the buffer into a message data object which can manipulated by the other library routines. The following table describes the C library routines for message packaging.

`void object_create(name)`
`char *name;`
This routine creates a new message data object in the word of objects. The ***name*** argument is a case sensitive ASCII string which will be used to reference the object. This name must begin with a letter.

`void object_rm(name)`
`char *name;`
This routine removes the message data objec,t referred by ***name***, from the world of objects. If no such object exists, the routine does nothing.

Table 24: C Library Routines for Message Packaging

1. The binary buffer format is a more compact representation of the message data object, however, the use of the floating point data types (OB_FLOAT and OB_DOUBLE) within the binary buffer format is not guaranteed to work across platforms. The C library routines for message packaging support the binary buffer format, but the Tcl library routines do not support this format since Tcl cannot handle binary data. As a result when integrating C programs with Tcl programs, the ASCII buffer format should be used. The specification of both formats is supplied in an appendix.

```
int object_mv(name1, name2)
char *name1, *name2;
```

This routine renames a message data object frome ***name1*** to ***name2***. If an object of ***name1*** does not exist, the routine will return -1. Otherwise the routine will return 0. If an object of ***name2*** already exists in the world of objects, it will be overwritten.

```
int object_exists(name)
char *name;
```

This routine tests whether the object refered to by ***name*** exists in the world of object. It returns 1 if it does exist, and 0 otherwise.

```
char **objects(argc)
int *argc;
```

This routine routine returns a list of all objects in the world. The returned list is an array of character strings (*argv* style). If there are no objects in the world, then an empty list will be returned. The ***argc*** argument is a pointer to a pre-allocated integer. If it is not NULL, then it will be filled in with the number of elements in the returned list. The returned list will be overwritten on successive calls to the **objects** routine. If it is desired to save the list, then it must be copied before the next call to **objects**.

```
void object_slot_set(name, slot, value, type, length)
char *name, *slot;
void *value;
int type, length;
```

This routine sets a slot within the object refererred to by the ***name*** argument. If the object does not already exist, it will be created. The ***slot*** argument is a case sensitive ASCII string (starting with a letter) which will be used to refer to the slot. The ***value*** argument is a pointer to data which will be copied into the slot. This data is a single or array of one of the types described in table 23. The ***type*** argument specifies the type of the data and is the integer type value. These type values are also defines as macros of the type names within the library header file. The ***length*** argument, is the number of elements of the specified ***type*** to which ***value*** points. For example, to set a slot to contain a vector of three signed long integers, the ***value*** argument would be a pointer to an array containing three **ints**, the ***type*** would be 9 (or *OB_SLONG*), and the ***length*** would be 3. This routine copies the data into the message object, so any further manupilation of the original value will not be reflected within the object.

```
void object_slot_rm(name, slot)
char *name, *slot;
```

This routine removes a slot refered by the ***slot*** argument, from the object refered by the ***name*** argument. If the object does not exist in the world, or the slot does not exist within the object, then the routine will do nothing.

```
void object_slot_mv(name, slot1, slot2)
char *name, *slot1, *slot2;
```

This routine renames the slot within the object refered by ***name***, from ***slot1*** to ***slot2***. If the object or slot does not exist, the routine will return -1, otherwise it will return 0. If the destination name refered by ***slot2*** already exists within the object, it will be overwritten.

Table 24: C Library Routines for Message Packaging

```
int object_slot_exists(name, slot, type, length)
char *name, *slot;
int *type, *length;
```

This routine tests whether the slot refered by the ***slot*** argument, exists within the object refered by the ***name*** argument. If either the object does not exist, or the slot does not exists, the routine will return 0, otherwise it will return 1. The ***type*** and ***length*** arguments are pointers to pre-allocated integers. If these arguments are not NULL and the slot exists, they will be filled in with the respective type and length of the data contained within the slot.

```
void *object_slot_get(name, slot, value)
char *name, *slot;
void *value;
```

This routine is used to get the data contained within a slot refered by the ***slot*** argument in the object refered by the ***name*** argument. The routine copies the data into pre-allocated space pointed to by the ***value*** argument. If the ***value*** argument is NULL, the routine will allocate space (using **malloc**) and the data will be copied into it. The return value is a pointer to this space. If either the object or slot does not exist, then NULL will be returned. It is up to the user of this routine to manage the memory passed into the routine, or allocated by the routine.

```
char **object_slots(name, argc)
char *name;
int *argc;
```

This routine routine returns a list of all slots which exist within the object refered to by the ***name*** argument. The returned list is an array of character strings (*argv* style). If the object does not exist in the world or if the object contains no slots, then an empty list will be returned. The ***argc*** argument is a pointer to a pre-allocated integer. If it is not NULL, then it will be filled in with the number of elements in the returned list. The returned list will be overwritten on successive calls to the **object_slots** routine. If it is desired to save the list, then it must be copied before the next call to **object_slots**.

```
void object_print(name)
char *name;
```

This routine prints the object refered to by ***name*** to *stdout* in a human readable format. This routine is useful for debugging purposes and allows the slots within the object to be easily viewed.

```
char *ascbuf_from_object(name, size)
char *name;
int *size
```

This routine represents the object refered by name as a fomatted ASCII buffer of bytes. The routine returns a pointer to the character buffer containing the packaged message. This character buffer will be overwritten on successive calls to **ascbuf_from_object**. As a result it should be copied if it is desired to save it. The ***size*** argument is a pointer to a pre-allocated integer. If it is non NULL it will be filled with the size of the returned character buffer. This returned character buffer is always guaranteed to have an ASCII nul (0) character following it in memory. It is intended that this buffer can be written to a file, or written to a connected program via a socket. The buffer may then be converted back into a message data object.

Table 24: C Library Routines for Message Packaging

`char *binbuf_from_object(name, size)`
`char *name;`
`int *size`
This routine works exactly the same as **`ascbuf_from_object`**, except the returned character buffer will be in a more compact binary format.

`char *object_from_ascbuf(buf, name)`
`char *buf, *name;`
This routine takes a properly formatted ASCII character buffer, pointed to by ***buf***, and unpackages it into a new message data object. The newly formed message data object may keep its original name by supplying NULL as the ***name*** argument. In this case, a pointer to the original name will be returned by the routine. The space to which this pointer points will be overwritten on successive calls. The newly created message data object may be renamed by supplying the new name as the ***name*** argument. If an object with that name already exists, it will be overwritten with the new message data object.

`char *object_from_binbuf(buf, name)`
`char *buf, *name;`
This routine works exactly the same as **`object_from_ascbuf`**, except that it takes as an argument a pointer to a properly formattted binary buffer.

Table 24: C Library Routines for Message Packaging

5.2.1 A Message Packaging Example

The following code is a simple example of message packaging. There are three peices of

```
/* This is the code for a message packaging example */

#include "object.h"
#include <strings.h>
#include <stdio.h>

/* Here is some data which will be packaged */
char user[] = "John C. Carney";
int numbers[] = {31, 22, 45, 2};
int weight = 160;

main () {
   char *buf;
   int size;

   int weight2;

   object_create("new_ob");
   object_slot_set("new_ob", "USER", user, OB_ASCII, strlen(user)+1);
   object_slot_set("new_ob", "FAVORITE_NUMBERS", numbers, OB_SLONG, 4);
   object_slot_set("new_ob", "WEIGHT", &weight, OB_SLONG, 1);
   object_print("new_ob");

   /* convert object to a buffer */
   buf = ascbuf_from_object("new_ob", &size);
   /* At this point buf could be written to a file or a socket */
   printf("The buffer is:\n%s\n", buf);
```

Table 25: C Message Packaging Library Example

```
    /* create a new object from a buffer */
    object_from_ascbuf(buf, "new_ob2");
    object_print("new_ob2");

    /* query the "WEIGHT" slot from "new_ob2" */
    object_slot_get("new_ob2", "WEIGHT", &weight2);
    printf("The weight is %d\n", weight2);

    object_rm("new_ob");
    object_rm("new_ob2");
}
```

Table 25: C Message Packaging Library Example

data, an ASCII string, and array of five integers and a single integer. A mesage data objected, named "new_ob" is created and slots are set for each peice of data. The object is then printed and converted to an ASCII formatted buffer. At this point the buffer may be sent to a connected program or even written to a file. The buffer is then converted back into a message data object, named "new_ob2". This object is printed and a value is extracted from one of its slots. The value is printed, confirming that it is the same as the original value which was originally set. Finally, both objects are removed from the world and the program ends.

At this point it is worth re-emphasizing that the message packaging library provides a rich set of routines for creating messages which can contain slots of varying types and sizes. Once a programmer has access to such a library, the difficult part of creating an application which uses message passing is in deciding on the information and messages which should be sent and the actions taken to respond to such messages. The message packaging library attempts to relieve the programmer from the details of the message format by providing a simple interface for creating messages.

5.2.2 Nesting Message Objects

Although there is no direct support by the library routines for nesting message objects, this may be accomplished indirectly. A message object may be created and converted into an ASCII buffer. This buffer can then be set as the slot of another message object. This process allows objects to be nested arbitrarily, however it relies on the programmer to understand and manage the representation of the final object which is created.

5.2.3 Efficiency and Caching Issues

Since there may exist an arbitrary number of message data objects on the world, each

manipulation of an object requires the library routine to locate the object within the world. Internally, these objects are stored in a list. Although the internal details do not affect the programmer interface, it is useful to be aware of this when writing efficient code. Locating an object within the list may require the library routine to search the entire list in the worst case. However, any time a message data object is manipulated in any way, it will be re-linked to the head of list list. This creates a "most recently used" caching scheme, where the most recently manipulated objects will be linked toward the beginning of the list, making them more quickly located. As a result successive, or near successive, manipulations of a message data object will be very efficient.

5.3 Tcl Library Routines for Message Packaging

The Tcl library routines for message packaging are parrallel to the C library routines. The Tcl routines only support the ASCII formatted buffer and do not support the binary formatted buffer. The nesting and efficiency issues presented for the C library routines also apply in the Tcl case. The following table describes the available Tcl library routines for message packaging..

object_create *name* This routine creates a new message data object in the word of objects. The ***name*** argument is a case sensitive ASCII string which will be used to reference the object. This name must begin with a letter.
object_rm *name* This routine removes the message data object referred by ***name***, from the world of objects. If no such object exists, the routine does nothing.
object_mv *name1 name2* This routine renames a message data object frome ***name1*** to ***name2***. If an object of ***name1*** does not exist, a Tcl error will be raised. If an object of ***name2*** already exists in the world of objects, it will be overwritten.
object_exists *name* This routine tests whether the object refered to by ***name*** exists in the world of object. It returns 1 if it does exist, and 0 otherwise.
objects This routine returns a Tcl list of all objects in the world. If there are no objects in the world, then an empty list will be returned.

Table 26: Tcl Library Routines for Message Packaging

object_slot_set *name slot value type length* This routine sets a slot within the object refererred to by the ***name*** argument. If the object does not already exist, it will be created. The ***slot*** argument is a case sensitive ASCII string (starting with a letter) which will be used to refer to the slot. The ***value*** argument is single element or a Tcl list of elements containing the data to be stored in the slot. This data is of one of the types described in table 23. The ***type*** argument specifies the type of the data and is the integer type value. These type values are also contained within global variables corresponding with the name of the type (to be used as a macro). The ***length*** argument, is the number of elements of the specified ***type*** to which ***value*** points. For example, to set a slot to contain a list of three signed long integers, the ***value*** argument would be a list containing three ints (stored as strings in Tcl), the ***type*** would be 9 (or *$OB_SLONG*), and the ***length*** would be 3.
object_slot_rm *name slot* This routine removes a slot refered by the ***slot*** argument, from the object refered by the ***name*** argument. If the object does not exist in the world, or the slot does not exist within the object, then the routine will do nothing.
object_slot_mv *name slot1 slot2* This routine renames the slot within the object refered by ***name***, from ***slot1*** to ***slot2***. If the object or slot does not exist, the routine will rraise a Tcl error. If the destination name refered by ***slot2*** already exists within the object, it will be overwritten.
object_slot_exists *name slot ?type ?length??* This routine tests whether the slot refered by the ***slot*** argument exists within the object refered by the ***name*** argument. If either the object does not exist, or the slot does not exists, the routine will return 0, otherwise it will return 1. The optional ***type*** and ***length*** arguments are names of Tcl variables. If these are passed in as arguments, and the slot exists, they will be filled in with the respective type and length of the data contained within the slot.
object_slot_get *name slot* This routine is used to get the data contained within a slot refered by the ***slot*** argument in the object refered by the ***name*** argument. The data is returned as a Tcl list.
object_slots *name* This routine routine returns a Tcl list of all slots which exist within the object refered to by the ***name*** argument. If the object does not exist in the world a Tcl error will be raised.
object_print *name* This routine prints the object refered to by ***name*** to *stdout* in a human readable format. This routine is useful for debugging purposes and allows the slots within the object to be easily viewed.
ascbuf_from_object *name ?size?* This routine represents the object refered by ***name*** as a fomatted ASCII string containing the packaged message. The ***size*** argument is a name of a Tcl variable. If it is passed in then it will be filled with the size of the returned string. It is intended that this string can be written to a file, or written to a connected program via a socket. The string may then be converted back into a message data object. This routine will raise a Tcl error if the object to be converted does not exist in the world.

Table 26: Tcl Library Routines for Message Packaging

object_from_ascbuf *buf ?name?* This routine takes a properly formatted ASCII character string, ***buf***, and unpackages it into a new message data object. The newly formed message data object may keep its original name by not supplying the optional ***name*** argument. In this case the routine will return the object's name. The newly created message data object may be renamed by supplying the new name as the ***name*** argument. If an object with that name already exists, it will be overwritten with the new message data object.

Table 26: Tcl Library Routines for Message Packaging

5.4 Summary

This chapter has served to present the Massage Packaging Library, which is the final library within the CHAT suite. The library allows for message data objects to be created. Data of various typoes may be stored in an arbitrary number of slots within a message object. The object can then be converted to a character buffer, which has one of two specific message formats. The details of these message formats are left out of this chapter, as the emphasis has been placed on the interface routines for these creating and manipulating messages. In addition, a user may utilize all of the functionality of the library without knowing the details of the message formats. The intent of the Message Packaging Library has been to provide a simple interface to a rich message format. This library is independant, however, the messages generated by the library may be used in conjuntion with the Communications Handling Library.

Chapter 6

Exercising the Tools

In chapters 3, 4, and 5 the CHAT suite of libraries were presented. This chapter will discuss a factory display program which has been developed and integrated into CAFE, MIT's Computer Aided Fabrication Environment. Emphasis will be placed on the specific integration architecture which has been implemented. Finally, the chapter will conclude with a brief description of the integration of a graphical user interface and simulation environment with a run by run control server [Moy95], as well as other planned integration projects at MIT.

6.1 CAFE Overview

CAFE is a software system for use in the manufacture of integrated circuits, providing day-to-day support for both research and production facilities at MIT [McI92]. One task of the CAFE system is to maintain a database of lots and machines within a facility. Whenever a new lot enters the facility, or when any machine operation is performed, appropriate information will be entered into CAFE by a facility staff member. Multiple CAFEs may be running simultaneously, possibly over a network, with each operating on a common database.

6.2 Factory Display Overview

A factory display program, called *fdisplay*, has been developed. The *fdisplay* program is a

graphical program, written in Tcl/Tk, which displays a map of the semiconductor facility. Machines within the facility appear as icons on the display. The icon itself and the color of the icon may be used to display the status of a machine. Users may click on specific icons to determine which lots are currently in the processing queue at a machine. Users may also browse information about the various lots and machines within the facility. When a lot transfers from one machine to another, it may appear as an animated dot which moves along a path from the source machine to the destination machine. The following figure shows an example of the main window of the *fdisplay* program.

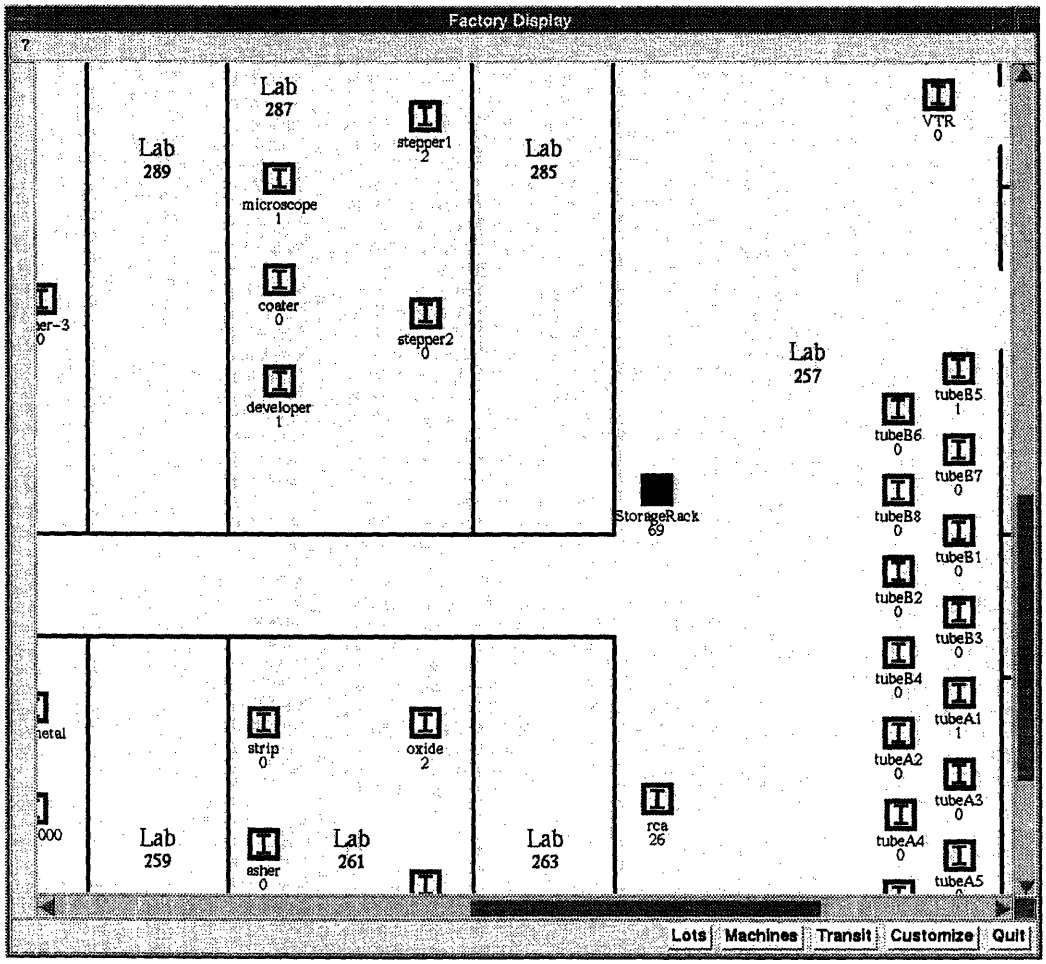


Figure 12: Example fdisplay Main Window

6.2.1 Controlling the Display

At the core of the *fdisplay* program is a command language which is used for controlling the display. The command language provides commands for configuring attributes of the

display, placement, manipulation and removal of objects within the display, as well as some utility commands. The idea is that the command language for controlling the display provides an interface for other programs which may wish to drive the display. It is intended that the *fdisplay* program will always be used in conjunction with a driver program. The fdisplay program is an output device, like a printer, and requires communication from a driving program to control the output. Detailed information about the *fdisplay* program and the display command language may be found in the appendix.

6.3 Factory Display Integration with CAFE

The idea of integrating the *fdisplay* program with CAFE, is to provide a real-time graphical display of the current status of all the lots and machines within the facility, with an active display that dynamically changes as the status information changes. The CAFE database contains this status information, however the information is continually changing as the facility staff enters new information into CAFE. This type of integration problem lends itself well to a message passing solution. As new status information gets entered into the CAFE program, CAFE may then send a message which contains the information to a running display. The display will then change to reflect the newly arrived message. This simple integration architecture is complicated by a few issues. The first is that multiple CAFE programs may be running simultaneously, possibly distributed over a network, with new status information continually being entered into each CAFE. Similarly, there may be several display programs being run simultaneously by many users of the system. New status information which is entered into any CAFE program must then cause a message to be "broadcasted" to each running display. The solution to both of these issues is with the use of a daemon program, called *fdaemon*. The *fdaemon* program runs continually at a centralized location and listens for connections from clients. Whenever a new *fdisplay* is started, it establishes a connection with the *fdaemon* program. At any given time, there may be several *fdisplays* connect to the *fdaemon* program. Whenever new status information is entered into any running CAFE program, CAFE establishes a connection with the *fdaemon* and then sends a message. The *fdaemon* program then distributes the message to all connected displays. The main job of the *fdaemon* program is to act as a gatherer and distributer of messages.

6.4 Sending Messages from CAFE

The CHAT suite is written in C and in Tcl/Tk, however CAFE is written in Lisp. As a result, CAFE cannot directly use the message passing libraries to send messages to the *fdaemon* program. CAFE does have the ability to spawn executable programs. A program, called *fmessage* was written in C and uses the CHAT libraries. At those specific locations within the CAFE code where it would be desired for CAFE to send a message, the *fmessage* program is spawned. The *fmessage* program then connects to the *fdaemon* program and sends the appropriate message. The idea of spawning a program from CAFE whenever a message needs to be sent may raise issues of efficiency. These messages occur infrequently, and the simplicity of the integration greatly outweighs any effort to make communication from CAFE more efficient.

6.5 Communication between *fdaemon* and *fdisplays*

Since the fdaemon program is written in C, and the fdisplay program is written in Tcl/Tk, these programs both use the CHAT libraries for message communication. The fdaemon program gets new status information from CAFE, it then converts this status information into a command in the fdisplay command language. The *fdaemon* then sends this command to each connected display. The *fdaemon* program is acting as the driver program for possibly several running *fdisplay* programs. Since the information and commands being sent are fairly simple, a message format of an ASCII string followed by a newline character was chosen as the message format throughout the entire integration. All message handling throughout the integration are done in an event based manner. The *fdaemon* program remains blocked until a new client connects or a new message arrives. Similarly, the *fdisplay* program remains blocked until a new message arrives.

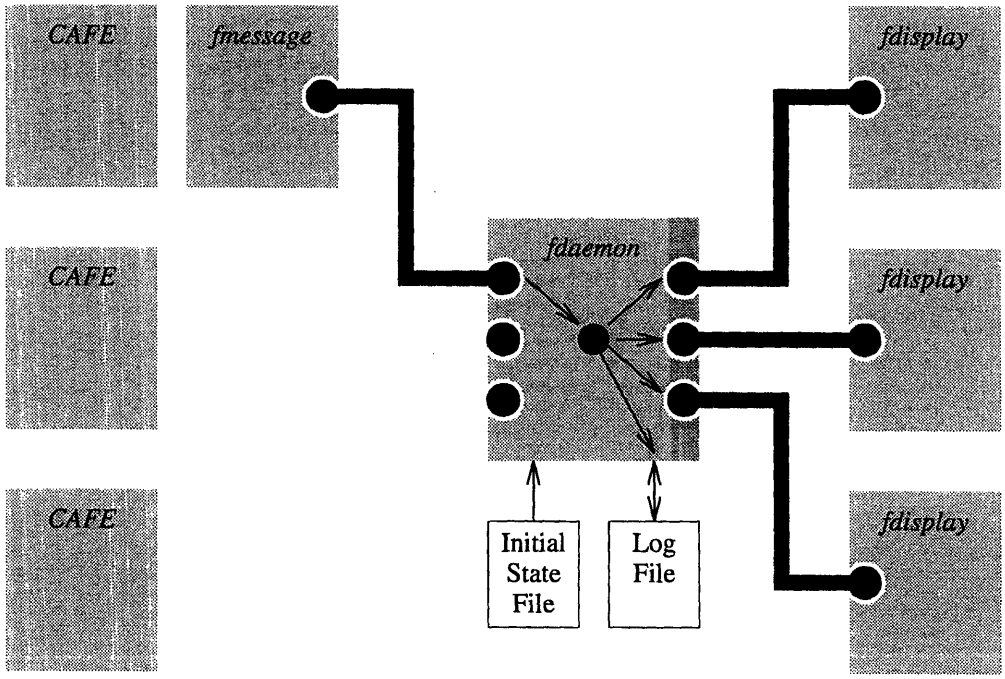
6.6 Initial State of the Display

Thus far, it only has been discussed what happens when new status information is entered into CAFE. This information ultimately causes messages to be sent and each running display to be updated. However, there must be a mechanism which gets the display into its initial state before it can handle these new status messages. Before the *fdaemon* program is run, an initial state file is extracted from the CAFE database. As new status information is gathered by the *fdaemon* program, it logs it to a file. Whenever a new display client con-

nects to the *fdaemon*, it is immediately sent the initial state file, along with the log file. This pre-loads the current state of the facility onto the display. This allows new display clients to connect in at any time. However, after long periods of facility activity the facility log file will grow, causing newly connecting clients to take longer to load up the current state of the facility. As a result, a daily or weekly maintenance schedule is executed whereby the log file is cleared and a new initial state file is generated from the CAFE database.

6.7 Integration Architecture

The integration architecture is shown in figure 13. In the figure there are three CAFEs and three *fdisplays* running, however, in practice these numbers are dynamically changing as users start up new CAFE sessions or new display programs.



Multiple CAFEs and fdisplays may be running simultaneously and distributed over a network. As new status information is entered into any CAFE, the fmessage program is spawned, causing a message to be sent to the fdaemon. This message then gets logged to a file and distributed to all connected displays.

Figure 13: CAFE / Factory Display Integration Architecture

6.8 Facility History and Facility Simulations

There is another use for the factory display program. The display can be used to "play-

back" facility events in much faster than real time. This would allow a user to perhaps view the entire previous week's event history in just a few minutes. The graphical manner in which a user can see all machines at every instant in time coveys an enormous amount of information in a very suitable way. While a view of the facility's historical events is useful, also very useful is a view of simulated facility events. The factory display software has been integrated with facility scheduling and simulation software to provide an "output device" for the data produced by these programs [Bon94].

The *fdriver* program has been developed and is written in C++. The job of the fdriver program is to read an input file containing a facility history (either real or simulated), and to drive an *fdisplay* program. It does so by sending messages at appropriate times such that the facility's history gets played out on the display much faster than real time. The *fdisplay* program also has the ability to send messages back to the *fdriver* program telling it to slow down, speed up, pause, and continue the play-back of facility events. The input file format for the *fdriver* program is very similar to the factory display command language and is described in the appendix.

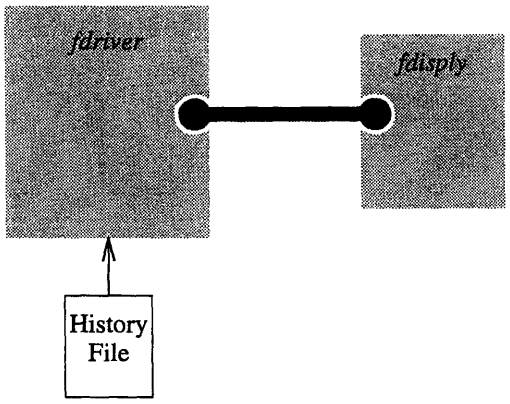
When a fdriver program is started, it automatically spawns an fdisplay program and a communication channel is established between the two programs. To a user of the program, this is transparent and it appears as though it is a single program. Figure 14 contains a diagram of the integration architecture.

6.9 Run by Run Control Integration

Libraries within the CHAT suite have been successfully used in the integration of a run by run control server with a graphical user interface and machine simulator[Moy95]. The run by run control server provides a computational engine which allows various statistical process control algorithms to be requested. The graphical user interface is a client program which connects to the server and contains a set of simulation, graphing, and archiving tools to aid in the test of the server. Other clients may also connect to the server, using the results of the server to set the control parameters on production equipment.

6.10 Other Integration Projects

There are other integration efforts in progress within MIT's Computer Integrated Design and Manufacturing (CIDM) project. Plans have been made to use the message passing



A file containing facility history (either real or simulated) is read by the fdriver program, causing messages to be sent to the fdisplay program at appropriate times.

Figure 14: Facility History / Factory Display Integration Architecture

libraries of the CHAT suite as part of this integration work. The remote fabrication project [Kwo95] and remote inspection projects [Kao95] are two such examples.

6.11 Summary

A factory display program has been developed and successfully integrated into CAFE by using the tools provided by the CHAT suite. This chapter has served to outline the architecture of the integration. Other integration efforts have successfully made use of the tools. Plans have been made to incorporate the tools onto other integration projects at MIT.

Chapter 7

Conclusion

Message passing provides an approach to software integration whereby structured information may be exchanged among running programs. The CHAT suite of libraries has been developed to provide flexible support for message passing within C and Tcl/Tk programs. Although the tools have been used successfully in integration projects, there are some limitations of the tools and potential for future work in this area. This chapter will address these issues.

7.1 A Graphical Tool for Software Integration

The libraries developed as part of the CHAT suite provide solutions for some of the lower level message passing issues. These include establishing connections, handling incoming messages, and packaging data into standard formatted messages. However, these libraries in a sense solve some of the easier, but nonetheless necessary, problems of software integration. Once we have established a mechanism for sending and receiving messages, the more difficult and higher level problems arise. What is the set of messages which may be sent or received? What are the events leading up to a particular message being sent? What actions are taken when a particular message is received? What is a suitable integration architecture? The answers to these questions will be unique for each integration project encountered. Although the Viewgraph program described in chapter 2, and the Factory Display program described in chapter 6 both are integrated using the message passing

tools, the integration architecture, messages and actions are very different.

A graphical model which describes the dynamic nature of the integrated application can be extremely helpful in designing working solutions. In chapter 2, the viewgraph example was modelled in this fashion. Useful as part of the future development of the message passing system would be a graphical tool. This tool could be used to aid in the modelling of integrated systems. Users would be able to schematically draw applications and connections as well as defining messages and events. When the integrated system is modelled the graphical tool would have the ability to synthesize the model into actual message passing code. The code would consist of function calls to the library routines within the CHAT suite. The resulting code may then be edited and added into new or existing applications to realize the integrated system.

7.2 Message Verification

As messages are received by the current suite of tools, they are parsed by a message parser to determine if a complete message has arrived. This restricts received messages to be of the proper format appropriate for the chosen parser. However, messages are sent by simply writing them to the connected socket descriptor without any restrictions on what may be written to a socket descriptor. As a result, it is possible to write a message which is not properly formatted. This may result in the buffer of received data to be corrupted, causing the parser to fail to parse any subsequent messages. To prevent this, it would be useful to provide support for message verification on the sending end of the connection. This would verify that all outgoing messages are properly formatted and that verified messages which are sent will be reliably received. Currently, this burden is placed on the programmer.

7.3 Time-out Mechanism

The message passing tools provide support for detecting when connections are dead. However, it is possible for a live connection to have a hung program connected to it. If this occurs, messages sent to that program will not be properly handled. It is possibly that the sending program is expecting a reply which will never come. In these cases it would be useful if the message passing tools supported a time-out mechanism. That is, if a reply message is not received within some amount of time, the program can perform some action to handle this exception. The action might be to simply assume the connected pro-

cess is hung and close the connection. This time-out functionality may be implemented external from the library routines, but a better solution would be to incorporate it directly into the libraries.

7.4 Efficient Message Formats

The message packaging library offers an interface to two message formats. There is a binary message format and an ASCII message format. The binary message format is a more compact representation of the data contained within the message. However, if the amount of data contained within the message is very large, then transmission of the message will not be very efficient. As a result, more efficient message formats can be designed. These message formats could use data compression algorithms to compress suitable messages. Efficiency will be gained when the CPU overhead for compressing and decompressing the message is less than the difference in transmission times between the larger and smaller versions of the message. An optimized system might have to measure data throughput and adaptively adjust its decision to compress or not based on this measurement.

7.5 Lightweight Version of the Libraries

The communications handling library allows programmers to specify a message format and appropriate message parser. There are many choices to choose from as well as a specification for designing a new message format and parser. However, the library could be simplified if it only supported a single message format. If one of the general purpose message formats were chosen as the "standard" message format, then a lightweight version of the library could be made based on that single format. The newly created library would not require the programmer to specify any information about the message format or message parser. However, the more flexible and heavier weight library would still be available if it were necessary to have the added flexibility.

7.6 Porting Libraries to Other Platforms

The libraries developed as part of the CHAT suite were written to run under Unix. It might be very desirable to run the libraries on other platforms which provide support for the TCP/IP protocols, such as personal computers. The library code uses some Unix specific system calls which must be modified when porting it to other operating systems. A version

has been successfully ported to IBM's OS/2 operating system. The OS/2 version of the libraries only provides support for TCP socket connections and does not support Unix domain socket connections. This does not remove any functionality from the libraries, however, socket connections on the same machine, if any, will have to be made using the less efficient TCP connection. The viewgraph server program described in chapter 2 was compiled to run on OS/2. Clients running on a Unix machine, successfully connected and communicated with the server. Future work may involve porting the libraries to more platforms.

7.7 Summary

As newer and larger software systems are being developed, software integration is becoming an increasingly important. An enormous effort is spent on integrating pieces of new software with existing software. There are many methods used to integrate software. Programs may operate on a common set of files, or on a common database. The focus of this thesis has been on using message passing as a mechanism for software integration.

Message passing allows communication of data and information among running programs. While there are some general purpose message passing systems available, they do not provide the flexibility to design message formats or specific integration architectures. As a result, the CHAT suite of libraries has been developed to provide a flexible level of message passing support for applications. The CHAT suite provides support for establishing connections, handling newly connecting clients and newly arrived messages, as well a support for packaging data into messages. It is intended that existing programs can be linked with the libraries and easily modified to include message passing capabilities. New programs can also be written to include these capabilities.

The CHAT suite has been written both in C and in Tcl/Tk, a scripting language used for building graphical interface. As a result a wide combination of C programs and Tcl/Tk programs may be integrated together to produce an integrated application. The viewgraph application, presented in chapter 2, and the factory display program, presented in chapter 6, are two examples where the message passing tools have been successfully used in software integration efforts. The tools have also been used in other software integration projects, and plans are underway to use them in other projects at MIT.

Although the message passing libraries are flexible, they serve the purpose of solving some of the lower level message passing issues. There is some room for future improvement of the libraries themselves, however, there is a significant room for future work aimed at solving some of the higher level issues of software integration. Each individual integration project will provide several new problems and challenges which must be solved. Future development of software design tools and frameworks aimed specifically at software integration will help to reduce the amount of pain and effort required to integrate software.

References

[Rei90] S. P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, July 1990, pp. 57-76.

[Dic94] A. Dickman, "The RPC-vs.-Messaging Debate: Under the Covers," *Open Systems Today*, August 15, 1994, pp. 58-59.

[Chu94] C. Chung, Y. Wang, W. Lin, Y. Kuo, G. Hsieh, "Tools Cooperation in an Integration Environment by Message-passing Mechanism," *Proceedings Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 94)*, Taipei, Taiwan, Nov. 9-11, 1994.

[Lib94] D. Libes, "X Wrappers for Non-Graphic Interactive Programs," *Proceedings of Xhibition 94*, San Jose, California, June 20-24, 1994.

[Sun91] SunSoft Inc, "The ToolTalk Service," *A SunSoft White Paper*, Revision 01, June 1991.

[Zim91] D.. Zimmerman, *The Finger User Information Protocol*, Network Working Group, RFC 1288, December 1991.

[Ous94] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, Reading, MA, 1994.

[McI92] M. B. McIlrath, D. E. Troxel, M. L. Heytons, P. Penfield, Jr., D. Boning, R. Jayavant, "CAFE - The MIT Computer-Aided Fabrication Environment," *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, Vol. 15, No. 2, May 1992, p. 353.

[Moy95] W. P. Moyne, *Run by Run Control: Interfaces, Implementation, and Integra-*

tion, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1995.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Ste90] W. R. Stevens, *Unix Network Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[Ald92] Aldus Corporation, *TIFF™ Revision 6.0*, Aldus Developers Desk, Seattle, WA June 1992.

[Chi94] V. Chin, *The CAFE Layout Program*, S.B. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994.

[Bon94] A. Bonvik, *Introduction to the CAFE Scheduling and Simulation Testbed*, Massachusetts Institute of Technology Internal CIDM Memo Series, Memo 94-11, September, 1994.

[Kwo95] J. Kwon, *Remote Fabrication of Integrated Circuits*, Massachusetts Institute of Technology Internal CIDM Memo Series, Memo 95-3, March, 1995.

[Kao95] J. Kao, *Remote Microscope for Inspection of Integrated Circuits*, Massachusetts Institute of Technology Internal CIDM Memo Series, Memo 95-5, March, 1995.

Appendix A

Using the Library Source Code

The libraries which make up the CHAT suite are available in C and Tcl/Tk source code. This appendix describes the source code files and how to incorporate the libraries into new programs.

The C Source Files

The following table lists the C source code files and header files which make up the CHAT libraries. To incorporate one or more libraries into a program, the C source code for the libraries must be compiled and linked with the program. The corresponding header files must be included by the program. Once the header files are included, the library routines may be used throughout the program.

CHAT Library	Source File	Corresponding Include File
Connection Support	basic_sockets.c	basic_sockets.h
Communications Handling	com_handler.c	com_handler.h
Message Packaging	object.c	object.h

Table 27: C Library Source and Header Files

The Tcl/Tk Source Files

The Tcl/Tk libraries of the CHAT suite require that the version of Tcl/Tk include the

TclDP, TclX, and TclCOM extensions. A Tcl/Tk "wish" shell can be built with these extensions (and some others) with a single install of the GoldTk package available by anonymous ftp at ftp://mtl.mit.edu/pub/CIDM/goldtk/goldtk3.6g.tar.gz. The following table lists each CHAT library and its corresponding Tcl/Tk source code. To incorporate a library into a Tcl/Tk program, simply use the Tcl **`source`** command on the desired library source file.

CHAT Library	Source File
Connection Support	basic_sockets.tcl
Communications Handling	com_handler.tcl
Message Packaging	object.tcl

Table 28: Tcl Library Source Files

Appendix B

Viewgraph Source Code

This appendix contains the source code for the viewgraph server application described in chapter 2. It is intended that this source code will provide a skeleton for which other integrated applications may be built. The source code is commented and should be self-explanatory, however, refer back to chapters 3, 4, and 5 for descriptions of the actual library routines used within the code. The code for the viewgraph client has been omitted, as much of the code for the client is used for creating the user interface and is not related to message passing.

```
/* Copyright (c) 1995 Massachusetts Institute of Technology.
 * All rights reserved.
 * This software is supplied "as is," without any kind of
 * warranty whatsoever, but not limited to the implied
 * warranties of merchantability and fitness for a particular
 * purpose.  MIT makes no representations about the suitability
 * of this software for any purposes and does not promise to
 * support it in any way.
 * This software is not to be incorporated into a commercial
 * product, or access to it sold, without written permission
 * from MIT.
 *
 * This software may be copied or distributed to others within
 * your organization that are within the United States or Canada
 * so long as this copyright notice is included.  This software may
 * not be distributed outside the United States and Canada.
 *
 * Please send bug reports, fixes, or enhancements to
 * cafe@mtl.mit.edu
```

Table 29: Viewgraph Server Source Code

```
 * or Room 36-277, MIT, Cambridge, Mass.  02139.
 */

#ifndef lint
static char copyright[] =
"@(#) Copyright (c) 1995 Massachusetts Institute of Technology.\n\
 All rights reserved.\n";
#endif not lint

/* John C. Carney
 * Massachussetts Institute of Technology
 *
 * vgd.c  -  Viegraph Server Program
 * This program is started to create a viewgraph session. Arbitrary clients
 * may connect to the server. Messages are received and sent by the server to
 * accomplish the viewgraph functionality.
 */

/* The following includes all three header files for the CHAT suite */
#include "basic_sockets.h"
#include "com_handler.h"
#include "object.h"

/* Other includes, used mainly for the file descriptor set macros */
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>

fd_set clients;        /* The set of socket desciptors of the clients */
int control = -1;      /* The current controller (-1 means nobody) */
int num_pictures = 0;  /* The number of pictures stored in message objects */
int submit_sfd = -1;   /* The descriptor of the client who is currently
                          submitting a picture */

/* This is the handler which is calles when a client dies */
void died(sfd)
int sfd;
{
  char *buf; int i, size, cpid = -1;
  unsigned char tmp;
  com_unregister(sfd);     /* Unregister message handling on the socket */
  close(sfd);              /* Close the socket */
  FD_CLR(sfd, &clients);   /* Remove socket descriptor from set of clients */
  if (sfd == control) {    /* Did the dead client have control? */
    control = -1;
    /* set control object to nobody */
    object_slot_set("control","USERNAME", "nobody", OB_ASCII, 7);
    object_slot_set("control","HOSTNAME", "", OB_ASCII, 1);
    object_slot_set("control","PID", &cpid, OB_SLONG, 1);
    buf = ascbuf_from_object("control", &size);
    for (i = 0; i < 256; i++)     /* send control message to all clients */
      if (FD_ISSET(i, &clients))
        write(i, buf, strlen(buf));
    tmp = 0;
    /* set the point object to off */
    object_slot_set("point","STATE", &tmp, OB_BYTE, 1);
```

Table 29: Viewgraph Server Source Code

```
    object_slot_rm("point","COORD");
    for (i = 0; i < 256; i++)
      if (FD_ISSET(i, &clients))  /* send point message to all clients */
        write(i, buf, strlen(buf));
  }
  if (sfd = submit_sfd) { /* Was the dead client in the middle of submitting
*/
    submit_sfd = -1;      /* an image ? */
    tmp = 0;
    object_slot_set("servstat", "STATUS", &tmp, OB_BYTE, 1);
    buf = ascbuf_from_object("servstat", &size);
    for (i = 0; i < 256; i++)  /* send a status message to all clients */
      if (FD_ISSET(i, &clients))
        write(i, buf, strlen(buf));
  }
}

/* A client wishes to take control */
void take(sfd)
int sfd;
{
  char *buf; int i, size;
  if (control == -1) {   /* if nobody has control */
    control = sfd;
    object_mv("incoming", "control");
    object_slot_set("control", "COMMAND", "CONTROL", OB_ASCII, 8);
    buf = ascbuf_from_object("control", &size);
    for (i = 0; i < 256; i++)  /* send control message to all clients */
      if (FD_ISSET(i, &clients))
        write(i, buf, strlen(buf));
  }
}

/* A client wishes to give up control */
void give(sfd)
int sfd;
{
  char *buf; int i, size, cpid = -1;
  if (control != sfd) return;  /* test whether client even has control */
  control = -1;
  /* set control object to nobody */
  object_slot_set("control","USERNAME", "nobody", OB_ASCII, 7);
  object_slot_set("control","HOSTNAME", "", OB_ASCII, 1);
  object_slot_set("control","PID", &cpid, OB_SLONG, 1);
  buf = ascbuf_from_object("control", &size);
  for (i = 0; i < 256; i++) /* send control message to all clients */
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));
}

/* A client is about to send an image */
void presubmit(sfd) {
  char *buf; int i, size;
  unsigned char busy;
  submit_sfd = sfd;
  /* Tell all clients that the server is busy */
  busy = 1;
```

Table 29: Viewgraph Server Source Code

```
  object_slot_set("servstat", "STATUS", &busy, OB_BYTE, 1);
  buf = ascbuf_from_object("servstat", &size);
  for (i = 0; i < 256; i++)
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));
}

/* A client has submitted an image */
void submit(sfd)
int sfd;
{
  char newob[10];
  char *buf; int i, size;
  unsigned char busy;
  sprintf(newob, "pic%d", num_pictures++);
  /* save image by moving message object to another name */
  object_mv("incoming", newob);
  object_slot_set(newob, "COMMAND", "PICTURE", OB_ASCII, 8);
  buf = ascbuf_from_object(newob, &size);
  for (i = 0; i < 256; i++) /* send new picture object to all clients */
   if (FD_ISSET(i, &clients) && (i!=sfd))
     write(i, buf, strlen(buf));
  submit_sfd = -1;
  busy = 0;  /* send status "idle" message to all clients */
  object_slot_set("servstat", "STATUS", &busy, OB_BYTE, 1);
  buf = ascbuf_from_object("servstat", &size);
  for (i = 0; i < 256; i++)
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));
}

/* The controlling client has chosen an image to display */
void display (sfd)
int sfd;
{
  char *buf; int i, size;
  if (control != sfd) return; /* test whether the client has control */
  object_mv("incoming", "display");
  buf = ascbuf_from_object("display", &size);
  for (i = 0; i < 256; i++) /* send display message to all clients */
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));
}

/* The controlling client has changed the pointer*/
void point (sfd)
int sfd;
{
  char *buf; int i, size;
  if (control != sfd) return; /* test whether the client has control */
  object_mv("incoming", "point");
  buf = ascbuf_from_object("point", &size);
  for (i = 0; i < 256; i++) /* send point message to all clients */
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));
}
```

Table 29: Viewgraph Server Source Code

```
/* This handler is called whenever any message is received by a client
 * The approach is to convert the message buffer to an object, and rename
 * the object to "incoming". Then the "COMMAND" slot is tested to see what
 * kind of message was received. The appropriate routine is then called for
 * the incoming message. Each routine will then look at the "incoming" message
 * and act appropriately based on the contents of the message.
 */
void handler(message, size, sfd)
char *message;
int size, sfd;
{
  char command[20];
  object_from_ascbuf(message, "incoming"); /* convert to an object */
  object_slot_get("incoming", "COMMAND", command);
  if (strcmp(command, "PRESUBMIT") == 0)
    presubmit(sfd);
  else if (strcmp(command, "SUBMIT") == 0)
    submit(sfd);
  else if (strcmp(command, "TAKE") == 0)
    take(sfd);
  else if (strcmp(command, "GIVE") == 0)
    give(sfd);
  else if (strcmp(command, "DISPLAY") == 0)
    display(sfd);
  else if (strcmp(command, "POINT") == 0)
    point(sfd);
}

/* This routine is called whenever a new client connects to the listening
 * socket descriptor */
void new_client(sfd)
int sfd;
{
  char picob[10], *buf; int i, size, newsfd;
  unsigned char busy;
  newsfd = bs_accept(sfd);   /* accept connection */
  FD_SET(newsfd, &clients);  /* add socket descriptor to the set of clients */

  /* Tell all clients that the server is busy */
  busy = 1;
  object_slot_set("servstat", "STATUS", &busy, OB_BYTE, 1);
  buf = ascbuf_from_object("servstat", &size);
  for (i = 0; i < 256; i++)
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));

  /* Send the control message, picture messages, display message,
     point message to the new client */
  buf = ascbuf_from_object("control", &size);
  write(newsfd, buf, strlen(buf));
  for (i = 0; i < num_pictures; i++) {
    sprintf(picob, "pic%d", i);
    buf = ascbuf_from_object(picob, &size);
    write(newsfd, buf, strlen(buf));
  }
  buf = ascbuf_from_object("display", &size);
  write(newsfd, buf, strlen(buf));
```

Table 29: Viewgraph Server Source Code

```
  buf = ascbuf_from_object("point", &size);
  write(newsfd, buf, strlen(buf));

  /* Tell all clients that the server is idle */
  busy = 0;
  object_slot_set("servstat", "STATUS", &busy, OB_BYTE, 1);
  buf = ascbuf_from_object("servstat", &size);
  for (i = 0; i < 256; i++)
    if (FD_ISSET(i, &clients))
      write(i, buf, strlen(buf));

  /* register handling on new client */
  com_register_event(newsfd, ch_parse_fbrace, handler, died);
}

/* Here is the main. There are some objects which will be used to store the
 * state of the viewgraph program. These are initialized. Then a listening
 * socket is created and registered */
main (argc, argv)
int argc;
char **argv;
{
  int sfd, cpid = -1;
  unsigned char pstate = 0;

  /* There should be a single argument containing the port number */
  if (argc != 2) {
    fprintf(stderr, "usage: %s port\n", argv[0]);
    exit(1);
  }

  /* These objects will be used to store the state of the viewgraph
   * application */
  object_create("control");
  object_slot_set("control","COMMAND", "CONTROL", OB_ASCII, 8);
  object_slot_set("control","USERNAME", "nobody", OB_ASCII, 7);
  object_slot_set("control","HOSTNAME", "", OB_ASCII, 1);
  object_slot_set("control","PID", &cpid, OB_SLONG, 1);

  object_create("display");
  object_slot_set("display","COMMAND", "DISPLAY", OB_ASCII, 8);
  object_slot_set("display","NAME", "blank", OB_ASCII, 6);

  object_create("point");
  object_slot_set("point","COMMAND", "POINT", OB_ASCII, 6);
  object_slot_set("point","STATE", &pstate, OB_BYTE, 8);

  object_create("servstat");
  object_slot_set("servstat","COMMAND", "SERVSTAT", OB_ASCII, 9);

  /* Create a listening socket descriptor */
  if ((sfd = bs_tcp_listen(atoi(argv[1]))) < 0) {
    fprintf(stderr, "Could not listen to port %s\n", argv[1]);
    exit(1);
  }

  FD_ZERO(&clients);  /* initialize the set of clients to empty */
```

Table 29: Viewgraph Server Source Code

```
  com_register_listen(sfd, new_client); /* register the listening socket
                                         * for handling */
  while(1) pause();  /* pasue and wait for events to occur */
}
```

Table 29: Viewgraph Server Source Code

Appendix C

Message Formats

The message packaging library presented in chapter 5 provides an interface for creating message objects. These objects can be converted into a serial buffer of bytes. The value in the library is that it does not burden the programmer with the underlying details of the message format. The programmer simply uses the interface routines to create messages and to add and remove the slots (or entries) into those message. In this appendix, the underlying binary and ASCII message formats are specified.

The Binary Message Format

The first format to be described is a binary message format based on TIFF, a tag-based file format for storing and interchanging raster images. The binary message format will generalize some of the ideas set forth in the TIFF specification. To read more about TIFF please read the TIFF 6.0 specification (Aldus Corporation, Seattle, WA).

The binary message begins with a 16 byte *object file header* which points to an *object file descriptor* (OFD). The object file descriptor can contain an arbitrary amount of entries, where each entry points to a specific piece of data.

Object File Header

The binary message must begin with a 16-byte header which abides by the following.

Bytes 0-3: The byte ordering used within the message. These four bytes should be 49494949 HEX for little endian byte ordering (least significant byte comes first) for any type of multi-byte numeric data within the buffer. These four bytes should contain 4D4D4D4D HEX for big endian byte ordering (most significant byte comes first).

Bytes 4-7: The number 01AE5B HEX. This number is arbitrarily chosen but identifies the binary message format. The ordering of this number depends on bytes 0-3.

Bytes 8-11: The size of the binary message buffer in bytes.

Bytes 12-15: The offset (in bytes) to the 0th OFD. The offset refers to the location within the buffer relative to the beginning of the buffer (which is at location 0). All offsets must be integer divisible by 4, placing the pointed information on 4-byte word boundaries. Other than the word boundary constraint, these offsets may point anywhere in the buffer and it is up to the reader to follow the pointers.

Object File Directory

The first four bytes of the *object file directory* (OFD) contain the offset to the next OFD, if one exists. If there does not exist another OFD then these four bytes should be 0. The next four bytes of the OFD contain the number of directory entries contained within the OFD. Following this should be 16-bytes for each entry within the OFD.

The 16-byte OFD entry should abide by the following format:

Bytes 0-3: The Type field.

Bytes 4-7: The Count field.

Bytes 8-11: Offset to the Tag field.

Bytes 12-15: Offset to the Value field.

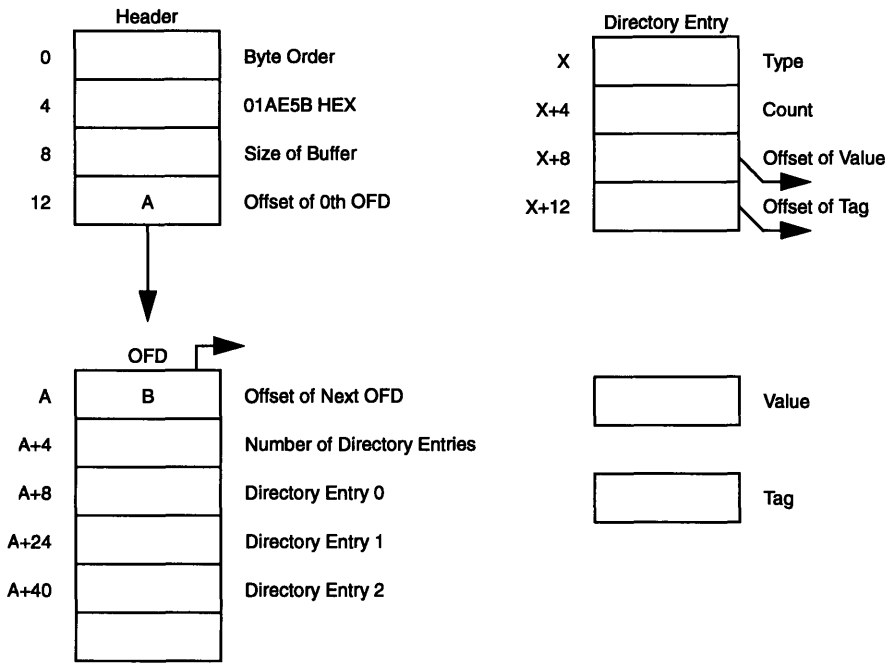


Figure 15: Binary Message Format

An entry contains an offset to a tag and an offset to it's associated value. The tag is simply a NUL terminated ASCII string. It should begin with a letter and contain only letters, numbers, and underscore characters. The value may be a single element of a particular type or a vector of elements of a particular type. The type field contains a a number which represents the type of data. The count field contains the number of elements of this type stored in the value field. Each entry is actually a one dimensional array of data, where a count of one is a single element.

Type	Description
1=BYTE	8-bit unsigned integer
2=ASCII	8-bit byte containing 7-bit ASCII code; the last byte must be NUL (0).
3=SHORT	16-bit (2-byte) unsigned integer.

Table 30: TIFF Types

Type	Description
4=LONG	32-bit (4-byte) unsigned integer.
5=RATIONAL	Two LONGs: the first represents the numerator of a fraction; the second, the denominator.
6=SBYTE	An 8-bit signed (twos-complement) integer.
7=UNDEFINED	An 8-bit byte that may contain anything.
8=SSHORT	A 16-bit (2-byte) signed (twos-complement) integer.
9=SLONG	A 32-bit (4-byte) signed (twos-complement) integer.
10=SRATIONAL	Two SLONGs: the first represents the numerator of a fraction; the second, the denominator.
11=FLOAT	Single precision (4-byte) IEEE format.
12=DOUBLE	Double precision (8-byte) IEEE format.

Table 30: TIFF Types

The ASCII Message Format

The ASCII message format is intended to be a mapping of the binary message format. It is advantageous to have both a binary message format and an ASCII message format. While it seems that the binary message format would be more efficient, some programming languages are better equipped to deal with ASCII messages.

The concepts of *object file descriptor* and *directory entries* will also be used in the ASCII message format. The entire ASCII message must be enclosed within braces, and each OFD must be enclosed within braces. Any type of white space (spaces and newlines) are allowed between successive braces. A typical message might have the following syntax:

{ {*0th OFD*} {*1st OFD*} . . . }

Note there is no *object file header.* It is not necessary in the ASCII message format. Each OFD should have the following syntax:

{*num_entries*} {*TAG TYPE COUNT* {*VALUE*}} {*TAG TYPE COUNT* {*VALUE*}} . . .

The number of directory entries within the OFD should be placed in the *num_entries* location. The *TAG, TYPE, COUNT* and *VALUE* locations are analogous to those in the binary message format. The following are some example entries:

An entry containing two shorts representing the dimensions of a matrix might look like the following. Note that individual numbers within the value are separated by white space. This is true of all the numeric types (1, 2, 4, 5, 6, 8, 9, 10, 11, 12).

```
{MAT_DIM 3 2 {12 15}}
```

An entry containing an ASCII string representing a street address might look like the following. Note here that the terminating NUL character is included in the *COUNT* (per the TIFF spec.) however this NUL character is not embedded within the ASCII buffer. Since it is possible for and ASCII string to contain curly braces which may interfere with the braces that define the message format. All curly braces within ASCII entries must be escaped by preceding it with the "\" character. Likewise, all "\" characters within an ASCII entry must also be escaped by preceding it with another "\" character.

```
{STREET_ADDRESS 2 17 {555 Memorial Dr.}}
```

An entry containing 32 bytes of UNDEFINED (type 7) data. While the data is actually 32 bytes, each byte is encoded with 2 ASCII bytes representing the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. These digits may have white space or newlines embedded within them.

```
{MY_ICON 7 32 {0AFF217C 30CA2B1A EC1A578A 0034DFA1}}
```

An entire message can be built up from entries. Here is an example of an entire message.

```
{ {{4} {name 2 15 {Carney, John C}}
       {dob 1 3 {11 1 71}}
       {height 11 1 {6.00}}
       {weight 11 1 {162.50}}}
  {{3} {favorite_food 2 6 {pizza}}
       {favorite_numbers 8 5 {-5 22 12 -8}}
       {favorite_icon 7 32 {0AFF2A7C 30CA2B1A EC1A578A 0034DFA1}}}
}
```

Interface with the Message Packaging Library

The message formats described may encapsulate multiple message objects into a single character buffer. However, the current version of the message packaging library only supports the conversion of single message objects into a character buffer. In this case, these buffers will only contain a single *object file descriptor*. Slots within the message object map directly into entries within the message. The name of the slot is stored in the *tag* field of the message. The *type*, *count*, and *value* fields of the entry are used to store the *type*, *length*, and *value* of a slot. The first entry of a message will always be used to store the name of the message object. The *tag* filed will always contain "_name", and the name itself will be of *type* 2 (ASCII) and stored within the *value* field of the first entry. This convention is used by the message packaging library, and object names will automatically be extracted from this entry within the message buffer.

Appendix D

Factory Display Details

The *fdisplay*, *fdaemon*, and *fdriver* programs were described in chapter 6. This appendix will provide a brief *fdisplay* user's manual, as well as a technical specification of the *fdisplay* command language and the *fdriver* file format.

User's Manual

The fdisplay program may be spawned from another program or started directly from the command line prompt. It is intended that fdisplay will be run in conjunction with a driver program via a socket connection. As a result, when the *fdisplay* program is launched it must be made aware of the socket address of the host driver program. The *-host* command line option followed by the host address can be used. The host address can either be in the form *ip_address*:*port*, for a TCP socket connection, or a unix path for a unix domain socket connection. If the *-host* option is not explicitly specified, then the *fdisplay* program will use the setting of the *FD_HOST* environment variable, if it has been set. The following lines are examples of how to launch the *fdisplay* program:

```
prompt% fdisplay -host garcon.mit.edu:1086
```

```
prompt% fdisplay -host /tmp/sock

prompt% fdisplay
```

Although these three examples are shown as lines being typed in at the command prompt, in many cases the driving program will transparently spawn the *fdisplay* program directly. Once launched the main window of the *fdisplay* program will appear. The window starts off "blank", until the driver program begins sending commands which will populate the display with a facility map including machines and lots. A detailed map can be loaded in seconds. The following figure shows an example populated main window

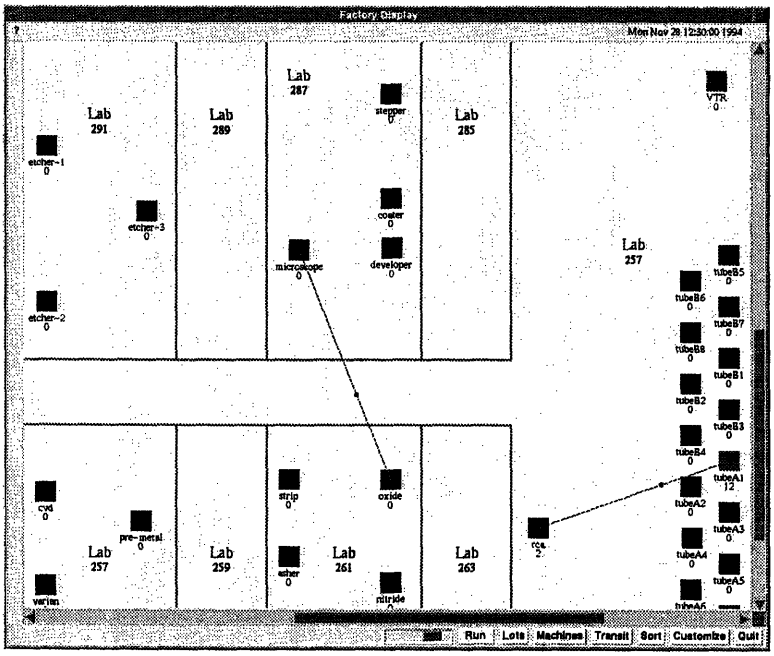


Figure 16: Populated Main Window

The main window is a resizable window which contains a scrollable canvas region along with a main tool-bar. Several objects are placed on the canvas. There are passive objects such as walls and two types of headings. The purpose of these objects is to provide the user a basic map of the facility. Although it is possible for these objects to be added or

removed during a display session, generally they will be placed at start-up time and remain static throughout the display session. The three other types of objects which may be placed on the display are machines, lots, and paths. Machines appear as icons on the display, with the machine name and number of lots directly below the icon. Lots may be placed at the various machines. As the lots are placed the number of lots indicators will change to reflect the newly placed lots. Lots may be removed from a machine and placed at another machine. During these transitions it might be desirable that the lot take some non-zero time to transfer between machines. The path object allows lots to be placed in a path and remain on the display while being transferred between machines. The path appears as a segmented line on the display and will only appear visible if it contains one or more lots. At various times a lot may appear as a dot on the display. The dot may move along a path to create an animation effect.

Obtaining Information From the Display

While the colors of the icons themselves on the display might provide the immediate status of a machine, the user also has the ability to probe the display to obtain various other status information about the machines, lots, and transit paths within the facility. At the bottom of the main window of the *fdisplay* program is a tool-bar containing several buttons. Three of these are labeled *Machines*, *Lots*, and *Transit*. Clicking the left mouse button on one of these will bring up a new window containing a list of all of the objects of the respective type within the facility. For example, clicking on the *Machines* button, will bring up a window containing a list of all the names of the machines contained within the facility. The names of the machines will appear in the color corresponding to the color of their icon on the display. Clicking on any machine name with the left mouse button will

cause a target to flash on top of the machine's icon on the display. This is useful as a quick way of locating machines within the facility. This is perhaps more useful when applied to lots. Clicking on the name of a lot with the left mouse button, will flash a target on the machine or transit path where the lot currently resides. Clicking the right mouse button on the name of an object within the list will bring up a hypertext-type window which contains any type of textual information about the object, as well as links to pictures, or executable programs. Clicking on picture link will allow the picture to viewed, while clicking on an executable link will spawn an executable program in the background. An example of a hypertext information window for a machine is shown in figure 17.

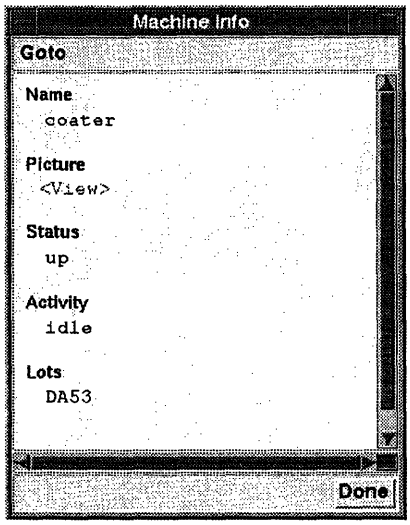


Figure 17: Hypertext Information Window

The hypertext information window may also be brought up by clicking the left mouse button directly on an icon for a machine, or a line representing a path.

In addition to information which is obtained from probing the display, at the top right of the main window is a 40 character text message which can be used to display any type of message to the user. For example, this message might be used in a real-time dis-

play to notify facility workers of urgent facility-wide status information.

Customizations

Another button on the bottom tool bar is the *Customize* button. Clicking on this will bring up the customize window. The customize window is shown in figure 18. The customize window contains a scale which allows the canvas of the display to be scaled. When sliding the scale bar, various parameters and toggles such as font and icon sizes will automatically be chosen. These sizes are the defaults for the desired scale of the canvas, but may be overridden by directly manipulating the entries within the customize window. After selecting the desired customizations, the *Apply* or *OK* button must be clicked for the canvas to be modified.

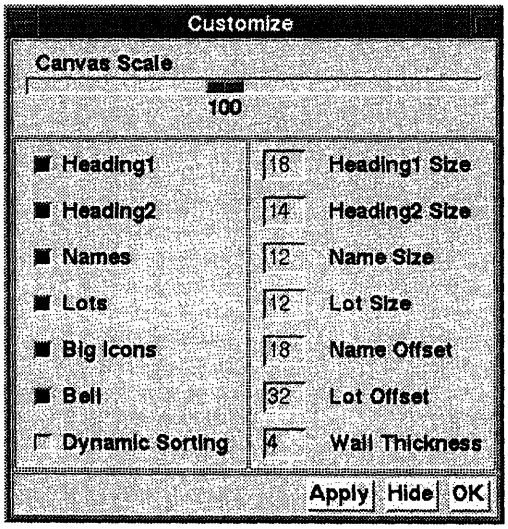


Figure 18: Customize Window

The customize window also contains a checkbutton to control whether the bell is audible. The bell is used as an alert to the user, but the user may turn it off as desired.

The final checkbutton in the customize window is the *Dynamic Sorting* checkbutton. The display maintains several lists. As commands are fed to the display the lists change. It is usually desirable to view these lists in sorted order. To do this the display

dynamically sorts each list as needed. When the display is heavily populated, this sorting may result in a sluggish operation of the display. As a result the user has the ability to control whether dynamic sorting is used. When dynamic sorting is off, an additional "Sort" button will appear in the bottom tool bar. This button can then be used by the user to sort lists whenever desired. The default initial state for dynamic sorting is on, however, the *-nds* (no dynamic sorting) command line option may be used when launching the *fdisplay* program to cause the initial state of dynamic sorting to be off.

The *fdisplay* Command Language

It is the responsibility of a driver program to feed commands to the *fdisplay* program over a TCP or Unix domain stream socket. The command language consists of several primitive commands which can be used by the driver program to manipulate the display. The command language syntax is that of the commands in the Tcl/Tk language. Several of the commands require arguments. These arguments may be lists. Lists in the Tcl/Tk language are usually enclosed in curly braces or double quotes, and elements within the lists are separated by white space. Any single arguments or list elements which contain white space should also be enclosed within curly braces or double quotes. Curly braces may be nested, but double quotes may not. Refer to a Tcl/Tk manual for further information about these rules.

The first commands which are fed to the display at start-up are configuration commands. These commands are used to configure the allowable attributes of objects, as well as the order in which they are displayed in the hypertext window, when the user desires more information about an object. There are several built-in attributes in addition to those configured by the user. These will become more apparent later when the object placement

commands are described. Each of the configuration commands should be sent at most once to the display and before any other commands.

The prototypes for the configuration commands are provided in the following table. Question marks are used to denote optional arguments. Bold is used for text exactly as it should appear in the command, and italicized text is used for supplied arguments.

setup_machine_attributes ?-*attribute default*? ?-*attribute default*? ... Defines additional attributes for machine objects. A default value is also associated with each attribute.
setup_lot_attributes ?-*attribute default*? ?-*attribute default*? ... Defines additional attributes for lot objects. A default value is associated with each attribute.
setup_path_attributes ?-*attribute default*? ?-*attribute default*? ... Defines additional attributes for path objects. A default value is associated with each attribute.
setup_machine_display ?-*attribute* {*heading type*}? ?-*attribute* {*heading type*}? ... Determines which attributes are to be displayed in the machine information hypertext window. Attributes are displayed in the order specified in this command. Associated with each attribute is a heading which is displayed in the hypertext window along with the value of the attribute. The type associated with each attribute must be one of: **text**, **picture**, or **executable**. If the attribute type is **text**, then the attribute value will be displayed as text in the hypertext window. For attributes of type **picture**, the value must be the path to a gif format image file, preceded by an @ symbol (i.e. `@/usr/pictures/etcher.gif`). For attributes of type **executable**, the value must be the name or path of an executable program. The following are built-in machine attributes and their types. These may also be specified to be displayed in the machine hypertext window. **Attribute** — **Type** name — text lots — text icon — text color — text
setup_lot_display ?-*attribute* {*heading type*}? ?-*attribute* {*heading type*}? ... Same as **setup_machine_display**, except for lot objects. The following are built-in lot attributes and their types. **Attribute** — **Type** name — text location — text color — text

Table 31: Configuration Prototypes

setup_path_display ?-*attribute* {*heading type*}? ?-*attribute* {*heading type*}? ...

Same as **setup_machine_display**, except for path objects.

The following are built-in path attributes and their types.

Attribute	Type
name	text
lots	text
color	text

Table 31: Configuration Prototypes

The next set of commands are those which are used for placement and removal of objects within the display. These objects include walls, headings, machines, lots and paths. In order to further manipulate objects placed on the display, a tag for each object is required as a reference to the object. A tag is simply a case-sensitive text string which begins with a letter and may contain any combination of letters and numerals. All tags must be unique in a global name space. The following table contains the prototypes for the object placement and removal commands.

place_wall *coord_list* ?**-tag** *wall_tag*?

Places a wall on the canvas. A wall is specified by a list of x and y coordinates where a straight line segment is drawn between adjacent coordinate pairs. The coordinates represent pixels in a Cartesian plane, with +x to the right and +y down. The coordinates need not be integer values. An optional tag can be specified. It is only necessary to specify this tag if it will be later desired that the wall be removed from the display.

place_heading1 *x y heading* ?**-tag** *heading1_tag*?

Places a large heading on the display at the specified coordinate. The heading is a text string. If it contains white space then it must be enclosed in curly braces or double quotes. A tag argument is optional.

place_heading2 *x y heading* ?**-tag** *heading2_tag*?

Places a small heading on the display at the specified coordinate. The heading is a text string. If it contains white space then it must be enclosed in curly braces or double quotes. A tag argument is optional.

Table 32: Object Placement and Removal Commands

place_machine *machine_tag x y name* ?**-icon** *icon*? ?**-color** *color*? ?*-flash period*? ?*-attribute value*? ... Places a machine on the display at the specified coordinate. A tag and machine name must be specified. Once specified, a machine's coordinate and name cannot be mutated. An icon for the machine may be optionally specified by providing a path to an X bitmap file preceded by an @ symbol (i.e. `@/bitmaps/etcher`). The bitmap specified should be of size 32x32 pixels, however it is also required that a smaller bitmap of 16x16 pixels exist in the same path and name, but with *_sm* appended to the name (i.e. the 16x16 bitmap `/bitmaps/etcher_sm` must also exist). If an icon bitmap is not specified then the default *block* icon will be used. The color for the icon can optionally be specified. The color defaults to black if not specified. The flash option allows the machines icon to flash every period millisecond. The default period is 0 (no flashing). Any of the additionally defined machine attributes may also be optionally specified. Any additional attributes which are specified, but not defined, will be ignored.
place_path *path_tag coord_list name* ?**-color** *color*? ?*-attribute value*? ... Places a path on the display. The path appears as a segmented line specified by a list of coordinates. The path will only appear visible when lots are in the path. A name for the path must also be specified. The coordinate list and path name cannot be mutated. A color for the path may optionally be specified. The color defaults to black if not specified. Any of the additionally defined path attributes may also be optionally specified. Any additional attributes which are specified, but not defined, will be ignored.
place_lots *lot_tag_list lot_name_list dest_tag* ?**-color** *color*? ?*-attribute value*? ... Places one or many lots on the display. Each tag in the list of lot tags is associated with the corresponding name in the lot name list. A tag must be specified as the destination of the lots. This destination tag may be that of a machine or a path. A color may be optionally specified. The color defaults to black if not specified. Any of the additionally defined lot attributes may also be optionally specified. Any additional attributes which are specified, but not defined, will be ignored.
remove_wall *wall_tag* **remove_heading1** *heading1_tag* **remove_heading2** *heading2_tag* **remove_machine** *machine_tag* **remove_path** *path_tag* **remove_lots** *lots_tag_list* Removes the respective objects from the display. When removing a machine or path from the display all lots residing at that machine or path will also be removed.

Table 32: Object Placement and Removal Commands

The next set of commands are those which deal with object manipulation. These commands allow for object attributes to be manipulated as well as for lots to be displayed and

moved. The prototypes for these commands are in the following table.

change_machine *machine_tag* ?**-icon** *icon*? ?**-color** *color*? *?-flash period?* *?-attribute value*? ... Changes one or many machine attributes. These attributes may include a machine's icon or color as well as any of the additionally defined attributes.
change_path *path_tag* ?**-color** *color*? ?*-attribute value*? ... Changes one or many path attributes. These attributes may include a path's color as well as any of the additionally defined attributes.
change_lots *lot_tag_list* ?**-color** *color*? ?*-attribute value*? ... Changes one or many lot attributes. These attributes may include a lot's color as well as any of the additionally defined attributes.
move_lots *lot_tag_list dest_tag* ?**-conditional** *cond_tag*? Moves all lots in the list of lots to the path or machine represented by the destination tag. If the conditional option and conditional tag are specified, then only the subset of lots residing at the machine or path represented by the conditional tag will be moved to the destination. By being provided with a conditional option, driver programs may possibly be simplified.
show_lots *lot_tag_list x y* ?**-conditional** *cond_tag*? Displays all lots in the lot list as a dot on the display at the specified coordinate. If the conditional option and conditional tag are specified, then only the subset of lots residing at the machine or path represented by the conditional tag will be displayed. This command can be used to create animation effects, where lots appear as moving dots on the display. Any subsequent **show_lots** command on a lot currently being display, will cause the lot to be redisplayed only at the newly specified coordinate. The dot will remain on the display until either a **move_lots** command or **hide_lots** command is performed on the corresponding lot.
hide_lots *lot_tag_list* ?**-conditional** *cond_tag*? All lots in the lot list which are being displayed as dots will be hidden. If the conditional option and conditional tag are specified, then only the subset of lots residing at the machine or path represented by the conditional tag will be hidden.

Table 33: Object Manipulation Commands

The final commands in the *fdisplay* command language are utility commands. These are described in the following table.

change_message *message* Displays a maximum of a 40 character message on the top right of the display.
ring_bell *message* Rings the bell on the terminal of the display. If the user has set the customization to turn the bell off then this command will do nothing.

Table 34: Utility Commands

load_file *filename* Executes the commands contained in the specified file. The commands in the specified file may be any of those in the command language.
set_flashing *state* If state is 1, then machine icon flshing is enabled for all machines whose flash attribute is non-zero. If state is zero, then flashsing is disabled for all machine icons.
set_auto_animate ***state*** *?transfer_time?* If state is 1, then the display is put in auto-animation mode. Every move_lots command will cause an animation effect where a lot appears to move along a straight line from the source machine to the destination machine. The *transfer_time* is the length of the animation effect in milliseconds. If *state* is 0, then the display is removed from auto-animation mode.
set_dynamic_sorting *state* If *state* is 1, then as activity occurs on the display, lists will automatically be sorted. If *state* is 0, then lists will not automatically be sorted. This creates a quicker display when running an animated facility simulation.
restart Removes all objects from the display. Initial set-up and configuration remains the same and the display may not be reconfigured.
kill Kills the current *fdisplay* program.
*line* Any line preceded by a pound symbol is considered a comment and ignored by the *fdisplay* program.

Table 34: Utility Commands

Socket Communication

Commands in the factory display command language are received as messages over a TCP or Unix domain socket connection. If the connected driver program should die, or if the socket connection is broken for any reason, the display will bring up a window notifying the user that the connection has been broken. All information up to that point will remain on the display and the display can be manipulated by the user in the normal way. However, new messages will no longer be received.

Outgoing Commands

While the display is viewed as an output device, it will send commands as messages over

the connected socket to the driving program. When the user presses the *Quit* button, the *exit* command will be sent over the socket to the driver program. The driver program can then use this information in an appropriate manner to stop operation. The other commands which are sent over the socket connection are those which can be used to control animation play and speed. In order for these commands to be sent, the factory display program must be launched with the *-controls* option. This will cause a speed control slide bar and a *Run/Pause* toggle button to appear within the tool bar. The *Run/Pause* toggle button initially starts in the paused state, with the word *Run* appearing within the button. If the user presses the *Run* button then the *run* command will be sent to the driver program. The button will then change states and the word *Pause* will appear within the button. If the user should press the *Pause* button, then the *pause* command is sent to the connected driver program. The button then toggles back to show the word *Run*. If the user slides the speed control slider bar, then the command *change_speed speed* will be sent when the user releases the slider bar. The *speed* is an integer number from 25 to 400 and the slide bar will always initially start in the 100 position. The following table summarizes the outgoing commands which are sent from the display program to the connected driver program.

exit This command is sent when the user presses the "Quit" button.
run This command is sent when the display is launched with the "-controls" option and the user presses the "Run" button. Pressing the "Run" button will also cause it to toggle to a "Pause" button.
pause This command is sent when the display is launched with the "-controls" option and the user presses the "Pause" button. Pressing the "Pause" button will also cause it to toggle to a "Run" button.

Table 35: Outgoing Commands

change_speed *speed*
This command is sent when the display is launched with the "-controls" option and the user releases the speed control slider bar. The *speed* is an integer value ranging from 25 to 400.

Table 35: Outgoing Commands

The *fdriver* Program

The *fdisplay* program described can receive commands to place objects and change their state on a display. The commands are executed as they are received. Historical or simulated facility data contains events occurring at various times. In order to display such information the *fdriver* program is used to send commands to a connected *fdisplay* program at the appropriate times, creating a dynamic display. The *fdriver* program reads an input file, processes the information contained in the file, spawns an *fdisplay* program, and awaits the *run* command from the *fdisplay* program. After receiving this command the *fdriver* program will begin sending commands to the *fdisplay* program over a unix domain socket. Between subsequent commands, the *fdriver* program will be idle for some time allowing for commands to be received by the display in a time sequence which is a scaled version of the actual time sequence for the real (or simulated) events. The default is for the driver to output commands 1000 times faster than the real (or simulated) event times.

Input File Format

Since the main use of the *fdriver* program is to "play" commands to the *fdisplay* program at the proper times, the input file format of the *fdriver* program contains mainly the same commands as those which are directly used to control the display itself. The organization of these commands within the file is important. The following figure shows the overall file

format.

```
display setup commands
initial state commands
begin_dynamic
dynamic commands
end_dynamic
```

Figure 19: *fdriver* Input File Format

The beginning of the file contains those commands which setup the display and attributes. The commands which follow are those which put the display into its initial state. These commands will all be sent to the display immediately after the file is processed. The initial state commands are then followed by the *begin_dynamic* directive, which informs the *fdriver* program that the following commands are dynamic commands. Each dynamic command must conform to the following prototype:

at *unix_time command*

The word "at" specifies that the event represented by *command* occurred at a particular date and time. The date and time is specified as a Unix time string. The Unix time string must be in double quotes and in the following format:

"*day month date hh:mm:ss year*" ,

example: "Mon Nov 28 16:20:35 1994"

The day and month are abbreviated and the hours are from 0 to 24. This time format corresponds to the format string "%a %h %d %T %Y" when used with the Unix commands contained in the *time.h* header file. The following table outlines which commands may be used in each section of the input file. The commands which are used are a subset of those described in the earlier section about the *fdisplay* command language. Refer back to that

section for information about particular commands.

Input File Section	Allowed Commands
display setup commands These commands setup the attributes of objects on the display as well as their ordering within the hypertext information window.	**setup_machine_attributes** **setup_lot_attributes** **setup_path_attributes** **setup_machine_display** **setup_lot_display** **setup_machine_display**
initial state commands These commands put the display into its initial state. This involves creating a facility map as well as populating it with machines and lots.	**place_wall** **place_heading1** **place_heading2** **place_machine** **place_lots** **place_path** **remove_wall** **remove_heading1** **remove_heading2** **remove_machine** **remove_lots** **remove_path** **change_machine** **change_lots** **change_path**
dynamic commands Commands are specified to occur at specified time. All dynamic commands must be preceded by the "**at** *unix_time*" syntax described earlier. It is **not** required that the dynamic commands appear in the file in sequential order by Unix time. Sorting will be done while the file is being processed.	**place_wall** **place_heading1** **place_heading2** **place_machine** **place_lots** **remove_wall** **remove_heading1** **remove_heading2** **remove_machine** **remove_lots** **remove_path** **change_machine** **change_lots** **change_path** **move_lots** **ring_bell**

Table 36: Allowable Commands in Input File

The **move_lots** command used by the *fdriver* program is slightly different than the **move_lots** command described earlier in the section on *fdisplay*. The following proto-

type should be used:

move_lots *lot_tag_list dest_tag* ?**-path** *path_tag tansfer_time*?

The command allows a path and transfer time in seconds to be specified with lot movement. If these are specified, the driver program will expand the move command into several primitive commands. This will effectively cause the lots to first be moved to the path. Then distributed over the path and transfer time, the lots will appear as dots on the path. Finally, when the lot reaches the end of the path it will be moved to the destination machine. If the file specifies that a lot be moved at any time while the lot is in transit, then the previous move will be preempted by the new move. For example, the *fdriver* program might expand the following command:

```
move_lots {lot1 lot2} etcher -path storage_to_etcher 1800
```

into the following sequence of primitives understood by the *fdisplay* program:

```
move_lots {lot1 lot2} storage_to_etcher
show_lots {lot1 lot2} 100 100 -conditional storage_to_etcher
show_lots {lot1 lot2} 100 110 -conditional storage_to_etcher
show_lots {lot1 lot2} 100 120 -conditional storage_to_etcher
show_lots {lot1 lot2} 100 130 -conditional storage_to_etcher
show_lots {lot1 lot2} 100 140 -conditional storage_to_etcher
show_lots {lot1 lot2} 100 150 -conditional storage_to_etcher
move_lots {lot1 lot2} etcher -conditional storage_to_etcher
```

These expanded commands are then "played" to the *fdisplay* program at the appropriate time to create the animation effect that a lot is moving on a path. The conditional construct in the *fdisplay* command language allows two or more *fdriver* **move_lots** to be performed on the same lots in an overlapping time interval and simply be expanded into primitive commands. As the commands are played to the *fdisplay* the most recent move will preempt the previous move. All moves may be animate without defining paths and using *-path* option. This is done by placing the *fdisplay* program in auto-animate mode.

Other Commands

At any point within the *fdriver* input file the **load_file** command may be used. This command has the following prototype:

load_file *filename*

This command is analogous to the *fdisplay* **load_file** command. Whenever this command appears within the input file the file specified will be included into the input file at that point. This command may be nested arbitrarily into the included files as well.

Any line of text within the input file preceded by a pound symbol is considered a comment and ignored by the program.

A Sample Input File

The following file is an input file for a single run of the *fdriver* program.

```
# demo file

# setup display
setup_machine_attributes -status "up"
setup_machine_display -name {"Name" text} -status {"Status" text} \
   -lots {"Lots" text}
setup_path_display -name {"Name" text} -lots {"Lots" text}
setup_lot_display -name {"Name" text} -location {"Location" text}

# put display into initial state (create map)
place_wall {60 200 20 200 20 20 300 20 300 200 260 200}
place_heading1 160 30 "Demo Lab"
place_heading2 65 170 "Rm. 100"
place_machine etcher 60 50 Etcher -color red -status down
place_machine tube 260 140 Tube -color green
place_lots {l1 l2} {Lot1 Lot2} etcher -color blue
place_path etcher_tube {60 50 260 140} "etcher -> tube" -color purple

# dynamic events
begin_dynamic
at "Mon Nov 28 08:40:00 1994" change_machine tube -status down -color red
at "Mon Nov 28 09:45:00 1994" change_machine etcher -status up -color green
at "Mon Nov 28 10:15:00 1994" move_lots l1 tube -path etcher_tube 1800
at "Mon Nov 28 10:30:00 1994" move_lots l2 tube -path etcher_tube 1800
at "Mon Nov 28 11:30:00 1994" change_machine tube -status up -color green
end_dynamic
```

Figure 20: Sample Input File

When "playing" the commands to the *fdisplay* program, the *fdriver* program will send commands uniformly in time to change the message in the upper right corner of the display. This message will display to the user the Unix time of the events as they are being

executed on the display.

The *fdaemon* Program

The *fdaemon* program accepts a single argument on the command line. That argument is the path to the *fdaemon* configuration file. The file sets variables which contain the port on which incoming messages will be received, as well as possibly multiple outgoing ports on which clients will connect. Associated with each outgoing port are initial state files and log files which may be set in the configuration file. As clients connect to the specific port, they will be sent these files as well as incoming messages as they occur. It is possible to specify filters for these files and the messages. The filters are simply regular expression substitutions and allow information to be transformed before it is sent to the client. For example in the CAFE integration, CAFE specific information is hard coded in the filters of the *fdaemon* configuration file, instead of coding it within CAFE or the *fdisplay* itself. This allows the fdaemon and fdisplay to be completely generic and configurable to the specific system into which it is integrated. The exact specification of the *fdaemon* configuration file is omitted from this brief overview, however, the commented sample configuration file supplied with the factory display distribution should be instructive in configuring the *fdaemon* program.