DISTRIBUTING A PLATFORM INDEPENDENT GRAPHICAL USER INTERFACE

Thomas Tilley

T.Tilley@gu.edu.au

GRIFFITH UNIVERSITY

GOLD COAST CAMPUS

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE HONOURS DEGREE OF BACHELOR OF INFORMATION TECHNOLOGY.

1998

Abstract

This dissertation details the design and implementation of a platform-independent graphical user interface framework. The framework presented uses distributed object technology to dynamically build an application's existing interface on a remote client. During the course of this work several distributed technologies were evaluated for their suitability in implementing the remote framework. This framework approach effectively provides Internet access to the application.

A dynamic interface implementation has a number of advantages over a "hard-coded" interface. Firstly, changes to the application do not necessitate changes to the remote framework. If the interface is modified in some way then the remote interface will automatically update itself to reflect those changes. No modification to the client software is required. Secondly, the remote interface can present the user with the expected "look and feel" as if the application were running locally.

Acknowledgments

There are a number of people I would like to thank who helped me through this project and my Honours year:

Firstly Professor Peter Eklund for his guidance and the opportunities he has afforded me.

I would also like to thank the Staff at MAPTEK for the assistance they have provided and for giving me the chance to play with their cool software.

I am eternally indebted to Simon Pollitt who was always there in my hour of technical need — often with scarily psychic answers to questions before I had even asked them.

Finally and most importantly I'd like to thank my Wife, Children and the rest of my Family for their love and support. Even when things were looking bad for our hero they still believed in me, graciously putting up with many late nights, long days and grumpy mornings.

Contents

List of Figures

List of Tables

List of Algorithms

Statement of Originality

The material presented in this dissertation has not been previously submitted for a degree or diploma in any university, and to the best of my knowledge contains no material previously published or written by another person except where due acknowledgment is made in the dissertation itself.

Thomas Alan Tilley

Chapter 1

Introduction

With the advent of portable computers many software applications are no longer confined to desktop machines in air-conditioned offices. The applications a professional uses can now be taken wherever and whenever they are needed. This option does not exist however for all applications. Some programs rely on large databases of information or expensive hardware that may restrict their portability.

One solution to the problem of physical portability is to provide an Internet "front-end" for these applications. With the appropriate software and hardware, users can access the application from a remote site.

Two main options exist for the implementation of a front-end for applications with graphical user interfaces (GUIs). The first is to "hard-code" a remote GUI specifically for the application and install it on the client users computer. The problem with this approach is the inflexibility of the interface. If the application is changed then the GUI software needs to be modified and re-distributed to the clients.

The alternative is to provide a GUI framework from which the application's interface can be built dynamically directly from the application server. This provides the remote GUI client with the same "look and feel" as the program running locally on the application server. An additional advantage is that changes to the back-end (server-side) application do not require modification of the front-end (client-side).

This dissertation details the design and implementation of a platform-independent GUI framework. The framework uses distributed object technology to dynamically build an application's existing interface on a remote client.

1.1 Aims

This work has two main aims. The first is to evaluate some of the distributed object technologies available to implement the objective. Having found an appropriate technology the second aim is to provide a working implementation of a platform-independent distributed GUI framework.

The scope of the project is broad. It incorporates client/server computing, distributed object technology, object oriented (OO) frameworks, and GUI design and implementation.

The architecture presented in Figure 1.1 represents the project's scientific hypothesis. The decision to accept or reject this hypothesis was dependent upon two factors. Firstly, was the architecture able to be realised within the time constraints imposed and, secondly, was the performance of such a system acceptable.

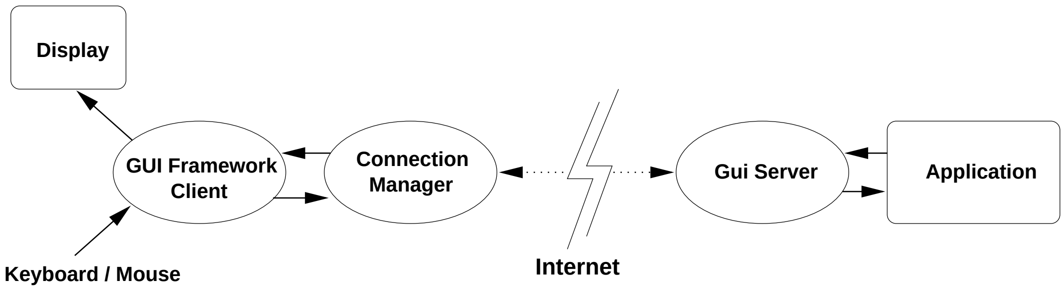

Figure 1.1: The project architecture.

Chapters 6 and 7 detail the design and implementation of this system. Chapter 6 discusses the Distributed GUI Server while Chapter 7 details the realisation of the client-side components.

The appropriate background material for distributed computing and GUIs is developed in Chapters 2 and 3 respectively.

Experimental results comparing the performance of four different distributed system implementations are presented in Chapter 4.

The remote application in the example implementation is a spatial database management system (SDBMS) named Carmen. Chapter 5 introduces Carmen and provides an overview of its architecture.

Chapter 8 summarises the dissertation, presents conclusions and indicates directions for further research.

The remainder of this Chapter briefly introduces two existing systems that serve as analogies for the project's architecture.

1.2 Background

There are a number of existing GUI systems built around a client/server architecture. While a general discussion of client/server computing is delayed until Chapter 2 the following sections introduce two distributed interface systems based upon this approach.

1.2.1 X-Windows System

The architecture of the X-Windows System [2, 7] is shown in Figure 1.2. An application's interface and the application itself are distributed over a network which is typically a Local Area Network (LAN). It is also possible to run both the application and its interface on the same machine and in such a case the distribution

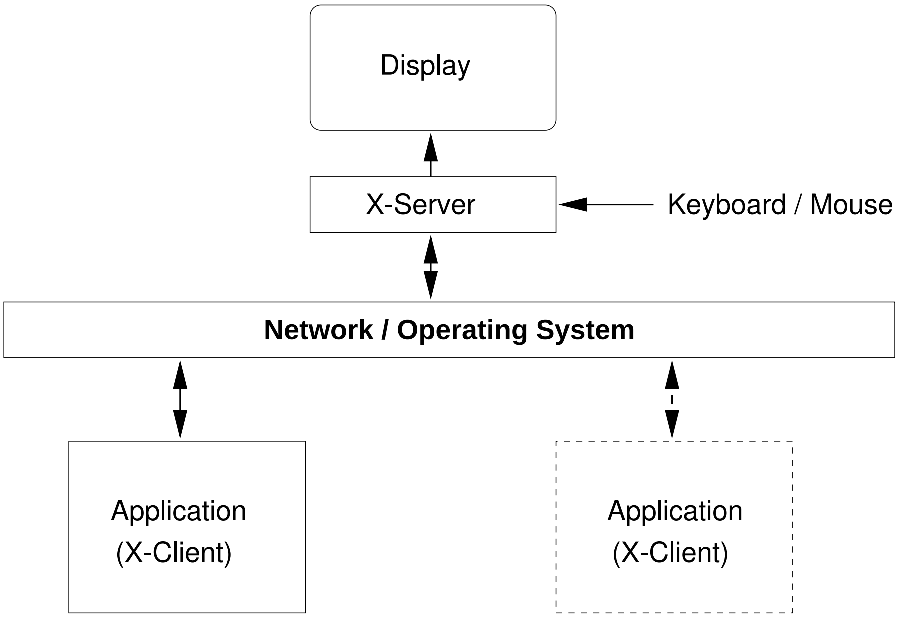

Figure 1.2: The X-Windows System.

is over the operating system. The application can be considered as the X-client and the GUI the X-server. The X-Server renders the GUI on the users display and handles input events typically from a mouse or keyboard. A number of applications can use this system concurrently. These applications may be executing on different computers within the network however their interfaces can all be implemented on a single X-server. The X-Windows system seamlessly abstracts over the fact that applications may be running on different machines.

The X-server in Figure 1.2 is analogous to the GUI framework client and connection manager in Figure 1.1. The network or operating system parallels the Internet and the X-client represents the application and GUI server.

1.2.2 Network PC

The Network Computer (NC), Network PC or NetPC provides another interesting analogy for this project. The three terms are technically different but are used interchangeably here as this discussion concerns the underlying principle of all three. The basic concept is to connect a computer with a minimal operating system and inexpensive hardware to a network. This stripped down computer is referred to as a "thin client". The application software for these computers runs on servers that are accessed via the network connection. The client computer serves purely to provide the interface for the application software [8]. All data storage and application computation takes place on remote servers. The client/server terminology used is more aligned to that of Figure 1.1 than the X-Windows example in Section 1.2.1. As with the previous example the idea is to separate an application's computation from

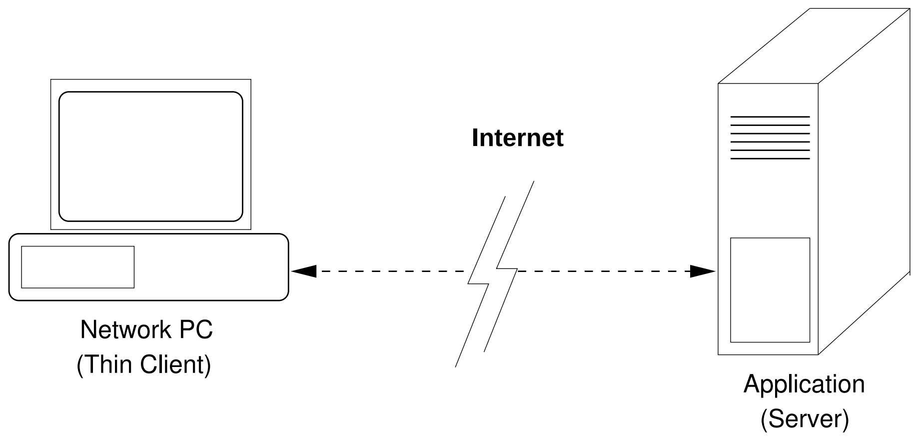

Figure 1.3: The Network PC concept.

its interface via a network.

Chapter 2

Distributed Computing

In this dissertation the term "distributed" typically refers to the simple separation of an application and its interface via a network. A more general definition of "distributed" relates to the distribution of processes or data over an unspecified number of machines. The distributed processes or data interact in a cooperative way as a single entity. In a computer network users are aware of the different machines connected to it and which machine is providing a specified service. Distributed systems on the other hand are transparent to the user. This chapter presents background material on distributed computing and explores some of the technologies used in its implementation.

2.1 Client/Server Computing

Distributed systems are typically based upon the client-server architecture represented in Figure 2.1

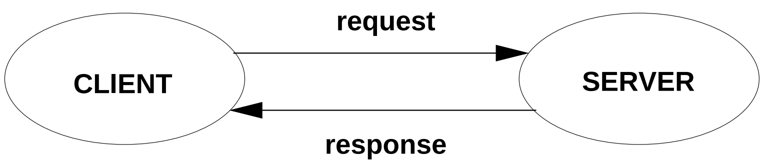

Figure 2.1: The client-server model.

The server process or computer provides services that can be used by the client. To make use of these services the client sends a request to the server. This request may contain parameters for the required service. The server processes the request and then returns a response to the client. This response contains any results produced by the requested service.

To relieve programmers from the complexities associated with making and receiving these requests a number of abstractions are introduced over the underlying communication mechanism. This underlying mechanism is typically based upon sockets.

2.2 Sockets

Sockets are a mechanism originally introduced with the Berkeley System Distribution of the Unix operating system (BSD Unix) that allow bi-directional communication between two computers. This is done by establishing a connection between the two machines through which they can then read or write from/to as if it were a file. A common analogy drawn in the literature is with the telephone system [25]. To receive telephone calls a person must firstly have a telephone installed and then be allocated a phone number. When a caller dials the phone number the telephone rings. The call is established when the callee picks up the handset. Communication can then occur in both directions. The telephone is analogous to the socket and the phone number to the socket's Internet Protocol (IP) address. Talking is analogous with writing to the socket and listening with reading from it.

Sockets are based on the concept of file interaction and so are largely concerned with the transmission and reception of characters. This is suitable for sending numbers, individual characters and strings but how do we transmit complex types or data structures? Serialisation provides the solution to this problem.

2.2.1 Serialisation

Serialisation is the process by which a complex structure or object is "flattened" so that it can be sent "down the wire". It essentially involves constructing a string representation for an object that can then be transmitted using sockets. Once the serialised representation has been received at the other end it can then be de-serialised to recover the object. Exactly how an object is serialised may have some bearing on the time taken to transmit and receive that object. A shorter string representation takes less time to transmit than a longer representation. In a system that sends and receives hundreds or thousands of objects this may have a significant effect on the systems performance.

Without serialisation communications between computers would be limited to simple integer and character types. Sridharan [25] uses the analogy of talking with people using the vocabulary of a three year old. While you can eventually convey your message it would be much easier to communicate if you had a larger vocabulary.

There are further issues to consider when using sockets to communicate between computers. Parameters required by a server must be serialised and assembled for transmission. Upon the completion of a service request any results must be disassembled and de-serialised. Marshalling is the process concerned with this packing and unpacking of parameters. The concept is analogous to railways where carriages are connected together in marshalling yards to form trains. The serialisation process is the sub-task of marshalling.

Sockets also need to contend with differences between how computers represent information. The byte ordering for integers differs between computers. Fortunately, there is a defined network byte ordering and standard operating system functions to convert from a host's byte ordering to network byte-order.

Sockets provide a general and reasonably flexible communication mechanism. However to achieve anything above simple messaging, application-level protocols need to be introduced. The design and implementation of protocols can be complex and error-prone [14]. An existing and commonly used protocol is the remote procedure call (RPC) described in the next section.

2.3 Remote Procedure Calls

The RPC is a widely used technique that underlies most distributed systems [26]. It is a protocol that implements the client-server model and abstracts over the low-level implementation details. For example programmers using RPCs do not need to provide the network address of the host that provides the services their client requires. RPC transparently handles the transportation of parameters to the server and the return of any results. While RPC is a protocol there are many different and incompatible implementations of it.

CORBA and Java's Remote Method Invocation (RMI) are two communication technologies that extend RPCs into the OO paradigm. These technologies are used for implementing systems containing distributed objects.

2.4 CORBA

CORBA is the Common Object Request Broker Architecture that is increasingly being used to distribute objects across networks. It provides a flexible communication and activation substrate for distributed heterogeneous environments [28, 29, 11]. CORBA is a key part of the Object Management Group's (OMG's) Object Management Architecture (OMA) It specifies the characteristics of the Object Request Broker (ORB) around which the OMA is built. The terms "client" and "server" become blurred when discussing CORBA. Each component in the OMA is both a client requesting services from other components and a server for the services it provides.

An ORB forwards requests for services to other ORBs and returns results to its client. The ORB abstracts over a number of important object attributes.

Firstly, the ORB hides the location of an object. Where the service request is being fulfilled is transparent to the client.

The execution state of an object is hidden by the ORB. If there is no object currently running to provide a service then the ORB will invoke one.

The ORB also hides an object's implementation. This is one of CORBA's biggest advantages. It enables a Java client for example to request a service which is implemented in C++. The client doesn't care how the service is implemented, it is only interested in the service provided. To facilitate this inter-language communication object interfaces are written in the OMG Interface Definition Language (IDL).

An interface represents a contract between the user of a service (traditionally the client) and the provider of a service (the server). It states what the client must provide in order to use the service (parameters) and what they will receive in return [25]. IDL language mappings currently exist for Ada, C, C++, Smalltalk and COBOL. The IDL also incorporates high-level abstractions like exceptions. Error handling is an important consideration in a distributed system [19]. One of the problems a programmer faces when implementing communication directly using sockets is the implementation of reliable error handling. How does the server inform the client that an error has occurred?

There are free ORB implementations available for a number of operating systems [1] including Java. Combining CORBA with Java applets (sometimes known as "ORBlets") provides users with transparent inter-operability. Users need never install CORBA on their machines as a Java ORB can be downloaded dynamically with the applet.

2.5 Remote Method Invocation (RMI)

Java's RMI is a CORBA like technology that facilitates distributed object communication between Java Virtual Machines [14]. An overview of the Java programming language is provided in Section 2.5.1.

As with CORBA remote objects in RMI must define interfaces for the methods that can be invoked on them. In CORBA the IDL is used to provide the interface then any language with an IDL mapping can be used to implement the object. In RMI both the interface and the implementation are written in Java. While CORBA provides a heterogeneous distributed environment RMI provides a homogeneous one.

Although RMI does not provide the flexibility of inter-language communication it does have advantages over CORBA. Firstly, because Java does not have to concern itself with a number of different languages it is far less complex in terms of usability. Once a remote object reference has been obtained it is treated exactly the same as an object in the local virtual machine. Apart from additional code to obtain the object reference the incorporation of RMI into Java programs is transparent.

Another advantage is Java's platform independence. While being restricted to Java may limit its suitability for legacy applications it does provide a highly-distributable format for new systems. Like CORBA, RMI includes the ability to throw remote exceptions and the forthcoming release of Java version 1.2 will include dynamic server invocation.

2.5.1 Java

Java is an OO language that was designed to be platform independent, web aware and secure. These web and platform independence attributes are highly desirable for

[1] http://adams.patriot.net/~tvalesky/freecorba.html

applications that require portability.

Platform Independence

Platform independence is essentially the ability to move software between different operating systems without re-compilation. This is reflected in the phrase "write once run anywhere". A program written in C needs to be compiled on each target platform before it can be executed. A Java program on the other hand needs only be compiled once. This produces Java "byte-code" which is then executed on a Java Virtual Machine (JVM). Java's portability is therefore restricted to those operating systems that support the JVM. This represents a large number of platforms [18].

Applets and Applications

Java Applets are small applications (hence the term "applet") that can be embedded in HTML pages [15]. An applet can then be accessed via the Web and run using a compatible web-browser. This technology is suitable for the implementation of small web accessible applications.

Downloading and running foreign code from the web is inherently dangerous. The potential exists for an application to intentionally damage a computer's file-system for example. To limit the use of applets for such malicious purposes they are restricted to running in an environment that is known as a "sandbox". The sandbox imposes security restrictions on the applet such as preventing it from reading or writing to the local file-system. While they provide security for applet users these restrictions also limit the functionality that applets provide and hence the applications they are suitable for.

Java can also be used to implement stand-alone applications. These applications run on the JVM and are free from the security restrictions imposed on applets.

Performance

Java's portability is based on the fact that it is an interpreted language. The price paid for this portability is performance. A Java program generally takes longer to execute than the same program written in C or C++. One approach to increasing Java's performance is to use Just In Time (JIT) compilers.

A JIT compiler converts the Java byte-code into target machine code and then saves the compiled code. If the same byte-code is called again, then rather than re-interpreting it the already compiled code is executed. This technique provides significant improvements in performance. When the second generation JVMs are introduced there should be a further increase in this performance [6].

Sun Microsystems new HotSpot virtual machine is like an optimising JIT compiler. It provides on-the-fly adaptive optimisation to produce "optimal" machine code for the target processor. While there is an initial penalty to pay for the extra time taken to perform optimisation any code that is re-executed should run more quickly. This results in an overall performance increase.

2.6 Performance in Distributed Systems

When considering the performance of distributed systems there are a large number of factors to consider. One of the most obvious is the difference between the invocation of a method on a local object and on a remote object. This difference is the latency of the two calls [30]. In terms of total execution time calls to remote objects are less efficient than local calls. As pointed out in Section 2.2 serialisation performance also effects this execution time.

In their paper on OO frameworks for distributed software, Schmidt and Fayad [23] warn software engineers not to rely on industry middleware standards like CORBA and RMI to reduce distributed software complexity. They suggest that although low-level implementations have reached maturity, the high-level services are not suited to real-time performance and reliability requirements. Therefore in Chapter 4 some empirical experiments are conducted to test the performance and timing results of distributed object technologies that are suitable for our objective.

Chapter 3

Graphical User Interfaces

The creation of a quality GUI is something of an art. It requires expertise in human factors, physiology, psychology and graphic design [24]. This chapter provides background material on this art. Section 3.1 firstly presents the MVC paradigm which is the basis of modern GUIs. The idea of Human Computer Interaction (HCI) is then introduced along with basic GUI concepts. The remaining sections discuss some of the tools and techniques available to implement GUIs.

3.1 Model View Controller (MVC)

In the early 1970's a research team at Xerox's Palo Alto Research Centre (PARC) set out to investigate effective human computer interaction. To implement the concepts they identified the OO language Smalltalk was developed [22, 4]. Modern GUIs were born out of the original ideas incorporated into the Smalltalk environment. The MVC framework they developed separates or decouples a GUI into three classes to increase flexibility and reuse: the Model class, the View class and the Controller class. The Model implements the application object, the View is its output representation and the Controller defines how the user interface reacts to input from the user. This concept is represented in Figure 3.1. Interface designs prior to MVC had lumped these objects together. Conceptually a GUI incorporates both the View and Controller which together interact with the application.

This approach has a number of useful advantages:

- the way a view responds to user input can be changed without modifying the view itself;
- when the model changes it can inform the view(s) so that they are updated automatically; and
- the implementation of any component can be changed without modification of any other objects that use it.

While the MVC model provides design abstraction there are other factors affecting GUI design that need to be considered.

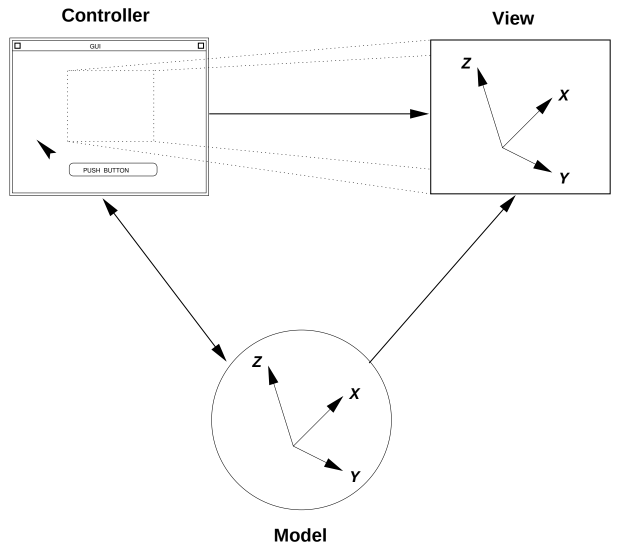

Figure 3.1: Smalltalk's Model View Controller (MVC).

3.2 Human Computer Interaction

HCI concerns itself with the human factors of software engineering [24]. Principles that consider how a user interacts with an application need to be incorporated into the design of a GUI. This effectively removes most HCI considerations from this project. The aim of this work is to reproduce an application's existing GUI irrespective of its "quality" in HCI terms.

There are however still a number of HCI issues to be considered. The design of the connection management and help windows detailed in Section 7.2 needs to incorporate good HCI principles. Response time and delay rate also become important because these interfaces form part of a distributed system. Shneiderman [24] presents a number of relevant response-time guidelines including:

- Users prefer shorter response times
- Longer response times (> 15 secs) are disruptive
- Response time should be appropriate to the task:
  - Typing, cursor motion, mouse selection: 50 - 150 milliseconds
  - Simple, frequent tasks: 1 second
  - Common tasks: 2 - 4 seconds
  - Complex tasks: 8 - 12 seconds
- Users should be advised of long delays

Having briefly explored some relevant design guidelines the following sections present basic interface implementation concepts.

3.2.1 Widgets

Widgets are basic GUI components that allow physical input devices (e.g. a mouse) to input a certain type of value (e.g. a button click) [20]. They typically include such items as menus, buttons, scroll bars and text entry fields. The fact that groups of widgets tend to have common base functionality makes GUIs excellent candidates for OO design. An example is the simple button hierarchy shown in Figure 3.2.

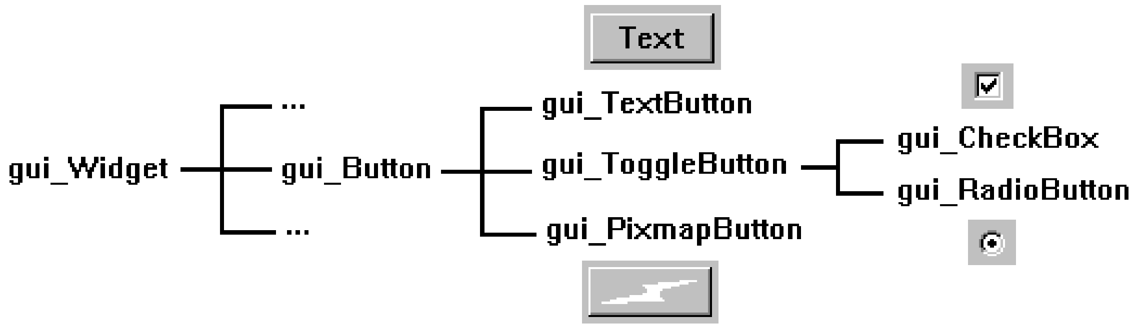

Figure 3.2: A simple widget hierarchy.

All of the buttons share the base widget functionality represented by "gui_Widget". Moving down the hierarchy the widgets become more specialised until the "leaves" of the widget tree represent all of the available button types. By substituting a different set of widgets it is possible to change the "look and feel" of a GUI.

3.2.2 Look and Feel

The general appearance and operational style of a particular GUI is known as its "look and feel". For example there are a number of window managers available for X-Windows, each with a different look and feel. The Macintosh and Microsoft Windows platforms each have their own distinctive look and feel. It not only involves the widget's look but how widgets are organised within a GUI and even their response to user input. Some software companies claim that copying their look and feel is a copyright infringement.

One reason for implementing the remote GUI as a framework is to support the look and feel of the application being accessed. If it was decided to change the position of two widgets inside a window then a hard-coded interface would need to be modified to reflect these changes. The dynamic approach provided by the interface framework means that no such changes are necessary. Provided the actual widgets are not modified no code changes or recompilations are required.

3.2.3 Callbacks

When a user interacts with a widget the interface needs to inform its application that an event has occurred. Event driven programming typically handles the user-input side of GUIs. In response to a pre-defined event (for example the click of a mouse button) some computation occurs. For an interface this computation is typically the invocation of a callback procedure. Callbacks are the responses produced by a GUI when a user operates a widget. Parameters may be included in a callback (for example a slider value) which are passed to the application.

Interfaces often contain hundreds of callbacks. In a distributed system these callbacks may need to be transmitted over the network to a remote application. Section 7.1 details how callbacks are implemented in this project's architecture.

3.3 GUI Building Tools

Most GUIs are built using software assistance from GUI toolkits. These provide existing widget sets and underlying functionality for their assembly and utilisation. There are also so-called Platform Independent GUIs available. The following sections explore the options for software assisted GUI implementation.

3.3.1 Application Frameworks

Much of the cost of software development is attributed to the continual "re-invention of the wheel" by software engineers. Being able to reuse software is seen as one way of reducing these development costs. Frameworks define generic components that can be reused to create new applications.

Fayad and Schmidt describe a framework as "a reusable, "semi-complete" application that can be specialized to produce custom applications" [3]. Both the Java Foundation classes and Microsoft Foundation Classes are examples of contemporary GUI frameworks. These frameworks provide a number of re-usable components that can be "assembled" to produce different applications. In the context of this project these are widgets assembled to produce different GUIs.

A number of benefits stemming from the use of OO application frameworks can be identified including modularity, reusability and extensibility.

Frameworks provide modularity through stable interfaces that abstract over implementation details. The interfaces partition the design and localise the effects of design changes. This is particularly important when designing an interface for a product under development where design changes are inevitable.

The stable interface of a framework can also be extended by applications to allow for new functionality via hook methods. Hook methods decouple the frameworks interface and application domain specific behaviour from context dependent variations of the application. In a GUI framework for example hook methods could be used to incorporate a new application specific widget.

In designing a framework those areas of the application domain that need to be kept flexible must be identified. These areas are called "hot spots" and a well designed framework should include a sufficient number [22]. Insufficient consideration of hot spots limits a frameworks re-usability.

The disadvantages of frameworks identified by Johnson [10] include;

- they are complex and therefore difficult to learn;
- they are hard to develop and therefore cost more and have a longer development time than normal applications;
- using a particular language to implement a framework limits its applicability only to systems supporting that language.

As pointed out in Section 2.5.1 Java is available on a large number of platforms. The choice of Java as a framework implementation language largely overcomes Johnson's final point.

Design Patterns

Frameworks typically include a number of design patterns. Design patterns are abstractions that represent rules to be followed rather than providing a concrete instantiation of some technique. Crane and Clyde [1] explored the use of design patterns to overcome some of the problems in developing quality GUIs. Their patterns attempt to capture expert knowledge in GUI design so that it can be reused in other interfaces. As already mentioned in Section 3.2 most GUI design issues fall outside the scope of this project. Crane and Clyde's work is mentioned here for completeness.

ET++

ET++ is a GUI application framework that builds upon lessons learned from MacApp - a popular application framework for the Apple Macintosh [31]. ET++ is implemented in C++ and is available in the public domain. Basic user interface building blocks (e.g. buttons), basic data structures and high-level application components are provided. The basic building blocks and the high-level components together define the look and feel of ET++ applications.

ET++ supports GUI development by taking care of such tasks as:

- window operations such as moving, resizing, activation and scrolling within windows;
- consistent file and dialog management across applications; and
- printer support for output in various formats.

ET++'s classes can also be adapted to provide sophisticated input event handling, text selection and manipulation, and clipboard support for pasting data between applications. These features make ET++ suitable for building text based applications such as graphical editors.

Current frameworks exist to implement a wide range of different software applications. However the earliest frameworks such as the MVC presented at the start of this chapter were used to support GUI implementation. The next section presents other software tools which assist in the construction of GUIs.

3.3.2 Toolkits

Myers [20] lists some of the problems associated with the development of user interface software as: the size of the code; the complexity of the code; difficulty in modification to the code; and difficulties in debugging the code. Software tools for GUIs attempt to address some of these problems. He argues that the use of interface building tools has many advantages including:

- the quality of the interface may be higher; and
- the code may be easier and more economical to create and maintain.

The use of third party software does however present a number of potential problems. Any bugs in the original code will filter down to the application under development. With commercial products the source code is not always available placing developers at the mercy of 3rd party customer support. This is a risk taken with any development environment but the complexity of user-interface software makes this problem more apparent.

Toolkits are widget libraries that can be called by application programs. Their limitation is that the widgets provided and their functionality may be limited and non-extensible. Although they have been designed to ease GUI programming they often have hundreds of procedures which can make them difficult to learn and use effectively.

Qt

Qt is a GUI software toolkit consisting of more than 140 C++ classes [27]. It runs on Unix and Windows platforms with a look and feel similar to the Motif toolkit used in X Windows. Qt's distinguishing feature is a unique "signal-slot" technique used to implement callbacks.

When a Qt widget changes state it emits a signal. Slots are used for receiving signals. By connecting a signal to a slot callbacks can be implemented. Multiple signals can be connected to a single slot and vice-versa.

Qt is an example of a toolkit that can be implemented on multiple platforms. To move a GUI built using Qt between platforms requires re-compilation of the software.

Swing

Swing is a platform-independent GUI toolkit implemented in Java. It is a major component of the Java Foundation Classes (JFC) [17]. Swing extends Java's existing Abstract Window Toolkit (AWT) to provide new widgets with increased functionality and flexibility.

The AWT relies on calls to the local operating system to render widgets on the screen. These operating system dependent widgets are known as "heavyweight" components. Swing on the other hand is implemented using mostly "lightweight" components. These are widgets implemented entirely in Java. This makes lightweight components extremely portable and allows them to have a "pluggable" look and feel (PLAF).

The PLAF can be used in two ways. Firstly, it allows an application to have the native look and feel of the platform it is running on. For example an application on a Macintosh will look and operate like a standard Macintosh application. If that application is then run on a Windows machine it will look like a Windows application. Alternatively the PLAF can be used to provide a fixed look and feel irrespective of the platform it is running on. At the moment Swing only implements two cross-platform look and feels — one for Motif and Swing's own "Metal" look and feel. The Macintosh and Windows look and feels are only available on their respective platforms for commercial reasons. Swing developers can however implement their own custom look and feels as required.

As always, there is a price to pay for this increased flexibility. The versatility provided by Swing comes at the expense of computation. Lightweight components generally take longer to re-draw than their heavyweight AWT counterparts.

There are other commercially available alternatives to Swing that either build upon or replace the AWT. The next section discusses the characteristics of so-called platform independent GUIs (PIGUIs).

3.3.3 Platform Independent GUIs

PIGUI toolkits provide software libraries that programmers use to produce GUI code for multiple platforms [12]. Functions and/or objects which are independent of the GUI being targeted are provided by the toolkit. For example to display a menu on the screen the PIGUI's menu function is called. When the code is compiled on a particular platform the PIGUI library displays that platforms menu on the screen in response to a menu function call. The idea is to provide a write-once piece of GUI code that works on multiple platforms. Unfortunately this approach has a number of disadvantages;

- PIGUI's slow down the execution of code;
- the resulting GUI is restricted to the features supported by a particular toolkit;
- PIGUI's generally target only a small number of platforms; and
- recompilation is required for each new platform.

PIGUI toolkits really provide a platform independent application programming interface (API) that targets platform dependent GUIs. While Qt does not suffer from all of these disadvantages, custom widgets can be created for example, it does require re-compilation on those platforms it supports. Both AWT and Swing solve the platform problems but AWT is not very extensible and Swing executes relatively slowly.

Chapter 4

Distributed Computing Analysis

This chapter describes a series of simple experiments conducted to evaluate the relative performance of different communication techniques. In addition to the empirical results obtained, the process of implementing the experiments provided valuable insight into the engineering costs associated with each technique.

4.1 Aim

The aim of these experiments was to gain some understanding of the relative performance of different communication techniques. Each of these techniques is a possible candidate for implementing distributed object communication within the GUI Framework.

A widget callback is representative of a typical "message" within the distributed GUI framework. This experiment's design attempts to provide a simple simulation of a widget callback. Performance is evaluated by measuring the total time taken for each technique to execute a number of these "callbacks". Section 4.2 describes the implementation of each technique.

4.2 Experimental Design

To simulate the invocation of a callback a string was passed from a simple client to an echo server which then returned the string to the client. This string represents a parameter being passed with the callback to the server. A program was written to generate files containing random strings with lengths between 1 and 256 characters. This program was used to generate ten files containing 10 strings and two additional files. One containing 20 strings and the other containing 40. This represents a total of 12 test files which were then used to evaluate the performance of the 4 techniques listed in Table 4.2.

Java was the language chosen to implement the client-side of the project's GUI framework because of its platform independence. Details of a target application around which the server-side of the project was implemented are presented in the next chapter. At this point it suffices to say that the target application is written in

Java Client	**C++ Server**	**Figure**
Sockets	Sockets	4.1
Sockets	Java Sockets/JNI	4.2
CORBA	CORBA	4.3
RMI	Java RMI/JNI	4.4

Table 4.1: Communication options between the Java Client and the C++ Server.

C++. To provide results with relevance to the final implementation of the project the experimental clients were implemented in Java and the servers in C++.

Four client-server implementations were evaluated in this experiment. The first implementation used sockets to communicate directly between Java and C++. This implementation is represented in Figure 4.1.

Figure 4.1: Java to C++ via Sockets.

Another sockets based implementation used Java sockets on both the client and the server. The Java Native Interface (JNI) was then used to inter-operate with C++. The JNI is a technique that allows Java to communicate via shared libraries with programs written in either C or C++ [13]. Figure 4.2 shows this alternative sockets implementation.

Figure 4.2: Java to Java via Sockets to C++ via JNI.

The pseudo-code representation of the sockets-based clients is presented in Algorithm 4.1. This algorithm takes the name of the test-file and the network address of the server as parameters. The random strings from the test file are firstly loaded into a vector data-structure. This is done to try and minimise the the influence of external factors — if the strings were read from the file as they were being transmitted some overhead may be induced by disk input/output. The client then connects to the server and once connection has been established the start time is recorded. Each string in the vector is then transmitted to the server and echoed back to the client. After the final string has been received the stop time is recorded and the elapsed time is calculated.

PROCEDURE TestClient(*Filename, Host*)
Load test strings from *Filename* into a *Vector* data structure.
Connect to the server *Host*.
Record the start time *Start*.
WHILE *Vector* contains more strings **LOOP**
    Transmit a string to the server *Host*.
    Receive the string back from the server *Host*.
END LOOP
Record the stop time *Stop*
Time = *Stop* - *Start*
Disconnect from *Host*.

Algorithm 4.1: Simple socket-based client algorithm.

The server algorithm for the sockets-implementations is represented by Algorithm 4.2. In an attempt to provide a realistic comparison between the techniques the Java to Java sockets implementation included a call to the JNI. This converted the string into C++ and then back into Java before it was returned to the client.

PROCEDURE EchoServer()
Accept a connection from *Client*.
LOOP
    Receive a string from *Client*.
    Transmit the string back to *Client*.
END LOOP

Algorithm 4.2: Simple Echo Server algorithm.

As discussed in Section 2.4 CORBA enables communication between Java and C++ (among other languages). The CORBA implementation represented a second aim for these experiments. The process of creating the different implementations provided a valuable insight into their engineering cost. That is the time and complexity of producing an implementation weighed against any advantages it may have. The CORBA implementation is represented by Figure 4.3.

The final implementation investigated again used JNI on the server-side but the client-server communication was implemented using RMI as shown in Figure 4.4. While CORBA and RMI are different technologies their client and server

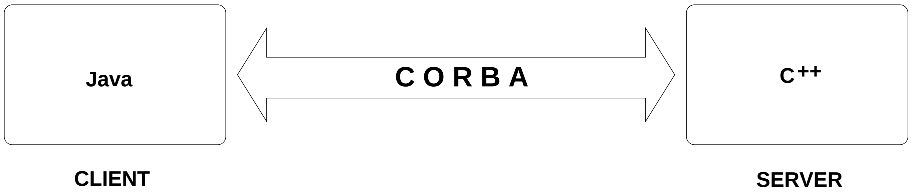

Figure 4.3: Java to C++ via CORBA.

implementations paralleled each other. Algorithm 4.3 provides an overview of the two distributed object clients.

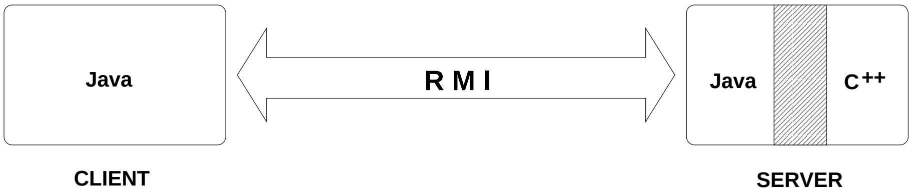

Figure 4.4: Java to Java via RMI to C++ via JNI.

PROCEDURE OOTestClient(*Filename, Host*)
Load test strings from *Filename* into a *Vector* data structure.
Obtain a reference to the object *EchoServer* on *Host*.
Record the start time *Start*.
WHILE *Vector* contains more strings **LOOP**
    Invoke the method *echo*(string) on *EchoServer*.
END LOOP
Record the stop time *Stop*
Time = *Stop* - *Start*

Algorithm 4.3: Simple distributed-object-based client algorithm.

As an alternative to providing a generic server-algorithm for the two distributed object implementations the CORBA IDL interface is listed in Algorithm 4.4. While RMI interfaces are actually implemented in Java the structure of the two is very similar. Within the RMI "EchoServer" a JNI call is made to convert the string into C++ and the back to Java. The code used is exactly the same as that of the Java to Java sockets implementation.

Having described the experiment's design Section 4.3 now presents the results obtained.

```
// EchoServer.idl
interface EchoServer
{
    string echo( in string name);
}
```

Algorithm 4.4: IDL interface for the CORBA "server".

Test	Java/C++ Sockets	Java Sockets	Java/C++ CORBA	Java RMI
1	8.1	17.3	32.5	32.8
2	7.0	14.7	29.5	34.4
3	6.4	13.7	22.8	27.7
4	8.6	15.4	25.9	31.1
5	6.1	16.6	22.4	30.5
6	5.2	13.2	23.5	24.4
7	5.8	14.2	26.1	30.5
8	5.8	15.3	23.3	33.2
9	6.1	13.7	22.4	27.5
10	6.2	15.3	25.6	35.7

Table 4.2: Average echo times (mS) for 10 strings.

4.3 Results

Each of the four client-server implementations were run using the pre-generated test files. Each of the 10 test files was run 10 times. The average time taken to echo the 10 strings for each implementation is presented in Table 4.3. The same data is represented by the graph in Figure 4.5.

The final two test-files containing 20 and 40 strings were also executed 10 times with each of the client-server implementations. Again both the client and the server were running on the same host. Table 4.3 incorporates these results with those from test-file 1 for comparison. These results are also represented by the graph in Figure 4.6.

The Java code was implemented using version 1.1.6 of the Java Development Kit (JDK) [1] for the IRIX platform. The CORBA code was implemented using Visibroker version 3.3 for Java and the IRIX version of Visibroker 3.2 for C++ under a 60 day evaluation license [2]. The test machine was a Silicon Graphics Origin 200 running IRIX 6.4.

[1] http://www.sgi.com/Support/about_surfzone.html
[2] http://www.inprise.com/visibroker/

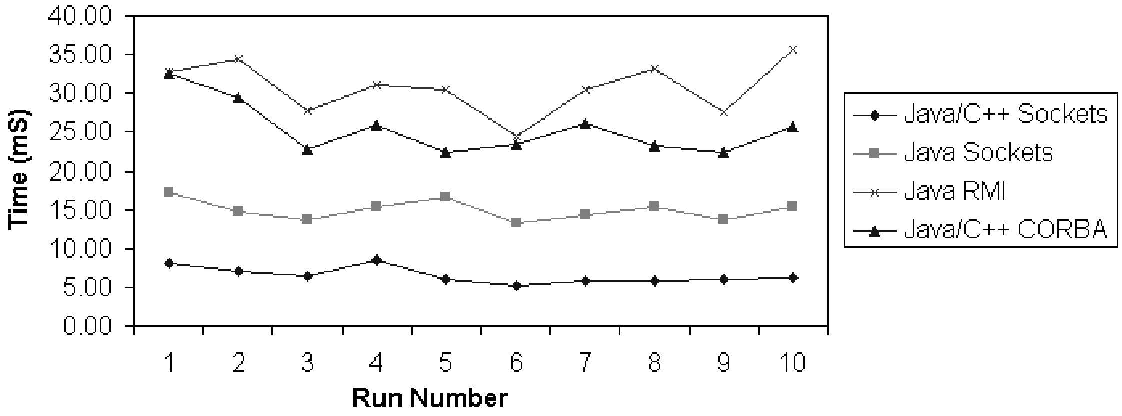

Figure 4.5: Average echo times (mS) for 10 strings.

No. Strings	Java/C++ Sockets	Java Sockets	Java/C++ CORBA	Java RMI
10	8.1	17.3	32.5	32.8
20	12.5	35.7	55.9	61.6
40	27.3	59.9	113.4	117.4

Table 4.3: Average echo times (mS) for 10, 20 and 40 strings.

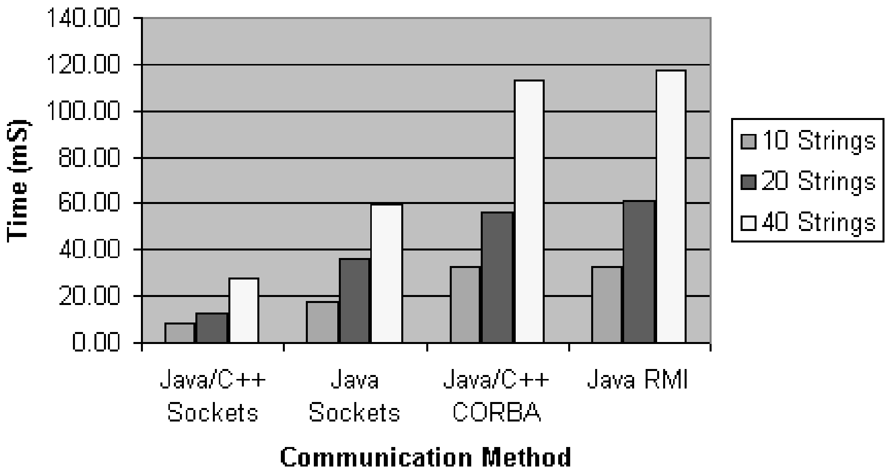

Figure 4.6: Average echo times (mS) for 10, 20 and 40 strings.

4.4 Interpretation

The results obtained echibit few surprises. While there is some timing variance in Figure 4.5 there is an obvious performance ordering on the basis of average echo time. Test files 6 and 3 were the smallest while files 4, 1, 8 and 10 were the largest. The overall shape of the graph is consistent with this information. Larger files obviously take longer to transmit.

The data in Table 4.3 and Figure 4.6 appear to indicate a linear increase in the time taken to transmit 10, 20 and 40 strings. Some initial experiments with 10, 100 and 1000 strings using the Sockets and RMI methods also reinforce this.

The remaining sections of this chapter address the performance of each technique in turn.

4.4.1 Java to C++ via Sockets

Not surprisingly the Java to C++ sockets implementation was the fastest of the four evaluated. Section 2.2 indicated that sockets are a fast, low-level implementation. While it appears to be approximately four times faster than RMI or CORBA with the test-data, using sockets directly in the distributed GUI implementation is unviable. The associated engineering costs to develop an appropriately high-level protocol given the time constraints imposed is too high.

4.4.2 Java to Java via Sockets to C++ via JNI

The Java to Java sockets implementation appears to take approximately twice as long to echo the same test-data as the Java to C++ method. The most likely cause is the conversion overhead introduced by the JNI. Using Java to Java sockets does provide some attractive functionality. Firstly, Java sockets effectively provide automatic serialisation. If a user creates a new object then provided they implement the "serializable" interface then Java sockets can ship it over the wire. The JNI also provides base type compatibility with C++ which helps address Java to C++ conversion issues. However, the JNI increases the complexity of the server and introduces some overhead. This technique still offers better performance in terms of speed than CORBA or RMI but again the engineering costs to implement a suitably high-level protocol for a distributed GUI implementation make it unviable.

4.4.3 Java to C++ via CORBA

The CORBA implementation is slower than the two sockets methods but it does have some significant advantages from an engineering point of view. The high-level functionality means that no custom protocol needs to be implemented. Remote exceptions can be incorporated to facilitate error handling and compatible base types for mapping between Java and C++ are provided. CORBA's big disadvantage in terms of this project is that it is complex to learn and use.

4.4.4 Java to Java via RMI to C++ via JNI

On the basis of the experimental results RMI is the slowest of the four implementations. It is also the last of the implementations to be examined. RMI's performance with this data, while slower, is close to that of CORBA. While RMI shares CORBA's high-level functionality and the ability to handle remote exceptions it does require JNI to inter-operate with C++. This introduces some performance overhead as well as increasing server complexity. While this implementation does not support "direct" Java to C++ communication we do acheive automatic serialisation between the client's JVM and the server's JVM. The JNI then provides compatible base types for C++ so the problems of type conversion can also be resolved. RMI however has one big advantage which is its transparent incorporation into Java applications. Apart from the addition of two lines of code to obtain a remote object reference an application's source code requires no change. Given the ever present time constraints imposed on a project of this sort,this makes RMI very attractive despite its performance.

Chapter 6 details the implementation of the Distributed GUI Server. For the reasons outlined above this server is implemented using RMI and the JNI. The next chapter introduces Carmen — the target application for a working distributed GUI framework.

Chapter 5

The Remote Application

The overall aim of this work is to enable remote access to an application. An implementation requires that we have a suitable application and the design of that application may have some bearing on the architecture used. The application presented here is a SDBMS called Carmen. SDBMSs are Database Management Systems (DBMS) that can store, retrieve, visualise and query spatial information. This chapter explores Carmen's design and its impact upon the GUI framework implementation.

5.1 Carmen

"Carmen" is the project name for a software product under active development by MAPTEK Pty Ltd [1]. Carmen is written in C++ and is designed to be "fully" Object Oriented. An important design aspect is its underlying client/server architecture which is represented in Figure 5.1. The ability to store spatial information and provide 2D and 3D visualisation make it suitable for mining, defence, urban-planning and 3D spatial information systems (SIS) applications. It is currently implemented on Silicon Graphics IRIX, DEC (now Compaq) Alpha and Microsoft Windows NT platforms.

Figure 5.1 represents a highly stylised view of the Carmen architecture. While this diagram conveys sub-cultural subtleties familiar to the Carmen software development team a more traditional representation of this architecture is depicted in Figure 5.2. An extensive discussion of the Carmen architecture is provided by Gibbs [5] [2]. His discussion is summarised in the following sections.

In the center of Figure 5.1 is the Master Control Process (MCP). This process handles all communication between the different components of the Carmen architecture, essentially acting as a packet switcher to route communication between the servers. Surrounding the MCP is the Data Engine which is the spatial database. The servers shown on the diagram can communicate with each other via the MCP

[1] http://www.maptek.com.au

[2] see http://www.int.gu.edu.au/kvo/reports/

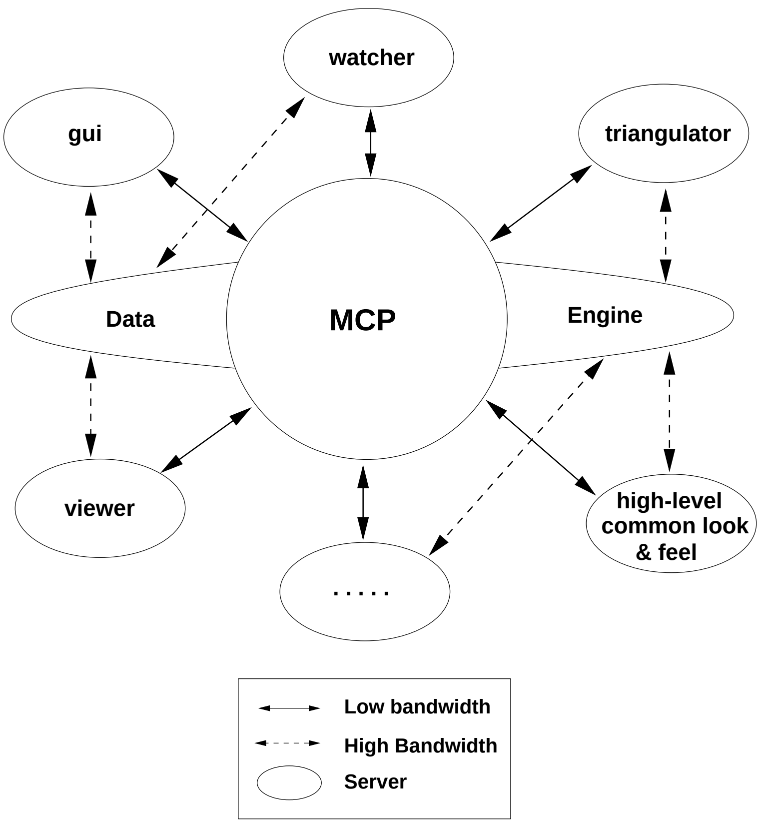

Figure 5.1: The Carmen architecture.

(the solid lines) and also directly with the Data Engine (the dashed lines).

This server-based architecture provides flexibility. A particular server only needs to be running when its services are required. This allows more effective utilisation of machine resources. It also means that new applications or services can easily be incorporated into Carmen by implementation as servers. The server labelled "..." represents any new application or service that could be incorporated into the architecture. In terms of this project this represents the Distributed GUI Server.

5.1.1 Master Control Process (MCP)

The MCP is the first process to be started whenever Carmen is run. As other servers start up they register with the MCP and identify the services they provide. When any server requires the services of another it sends its request to the MCP which invokes the appropriate function in the requested server. The MCP also handles the asynchronous passing of arguments such as filenames.

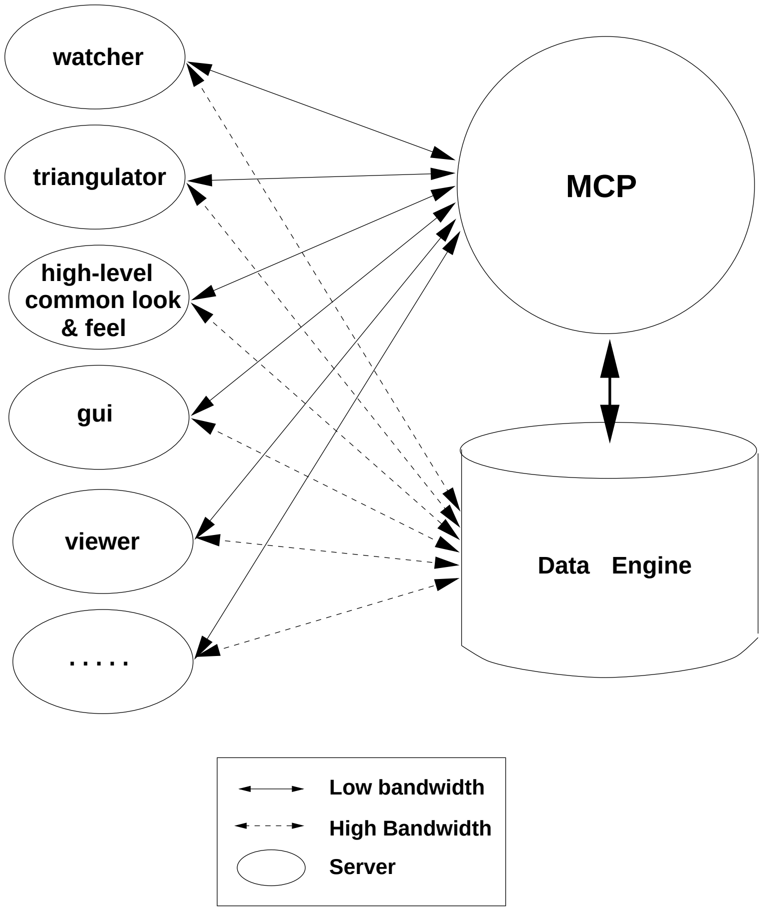

Figure 5.2: Alternative representation of the Carmen architecture.

5.1.2 Data Engine

The Data Engine is Carmen's persistent data repository. Within the Data Engine spatial information is stored in "slabs" analogous to files and "containers" analogous to directories. A container can store slabs or other containers to produce a hierarchical structure like the Unix file-system. Typical file-system conventions are used to navigate this hierarchy.

5.1.3 Watcher Server

The Watcher server handles database locking and concurrency issues in the Data Engine. It intelligently locks portions of the Data Engine in response to read/write requests by users or other processes. If data has been modified by one user or process then the Watcher can inform other processes interested in that data. If for example another process was providing a visualisation of the data its view could then be updated to reflect the changes.

5.1.4 High-level Common Look And Feel (HCLAF) Server

The HCLAF server provides different Carmen applications with a common "look and feel" (see Section 3.2.2). It controls the layout of widgets within a window so they are arranged in a consistent manner. This means that any new server added to the Carmen architecture automatically has a similar look and operational style to existing applications. It also provides a powerful programming abstraction by allowing requests without specifying how they are to be fulfilled. For example, an application may pop open a dialog window (via the GUI server) which can then be populated by a number of different methods. These population methods do not need to be explicitly stated by the application programmer.

Section 3.2 introduced the concept of HCI. The HCLAF server effectively puts HCI issues related to Carmen outside the scope of this project.

5.1.5 GUI Server

The GUI server is responsible for the creation, assembly and display of GUI components. This includes windows, dialog boxes and other widgets. The GUI server renders GUI components on the user's display according to instructions that come via the HCLAF server. It also handles user input passing callbacks via the MCP to the appropriate servers. The Qt toolkit described in Section 3.3.2 is used to implement Carmen's widgets.

An advantage of implementing GUIs using a separate server is that GUI updates are never held up by application procedures invoked by the interface. This is in fact true for all servers in this architecture. Any request for a service that does not require a response finishes as soon as the request is forwarded to the MCP. As an example suppose that a user selects a menu item that launches a new application. Once the GUI server has forwarded the request for the new application to the MCP it can then attend to other tasks. The GUI server does not need to wait for the new application to start.

5.1.6 Viewer Server

The Viewer server provides Carmen's 3D visualisation facilities. It provides the real-time 3D rendering of spatial data into windows. These "views" are managed by the GUI server which passes update requests (such as mouse-drags) to the Viewer server so it can respond appropriately. In terms of the MVC paradigm presented in Section 3.1 the information in the Data Engine represents the Model, the Viewer server is the View and the GUI server the Controller.

5.1.7 Master Application

When starting Carmen the user can optionally request which application they would like to use. This starts the application's server which then registers itself with the MCP. If no application is specified a default "Master Application" is started from which other applications can be launched. The Master Application's GUI is shown in Figure 5.3. It consists of a menubar, buttons, a slider, a status area and a view window. This view shows a 3D visualisation of a land basin. As described in the preceding sections the visualisation is handled by the Viewer server while the rest of the GUI is rendered and maintained by the GUI server.

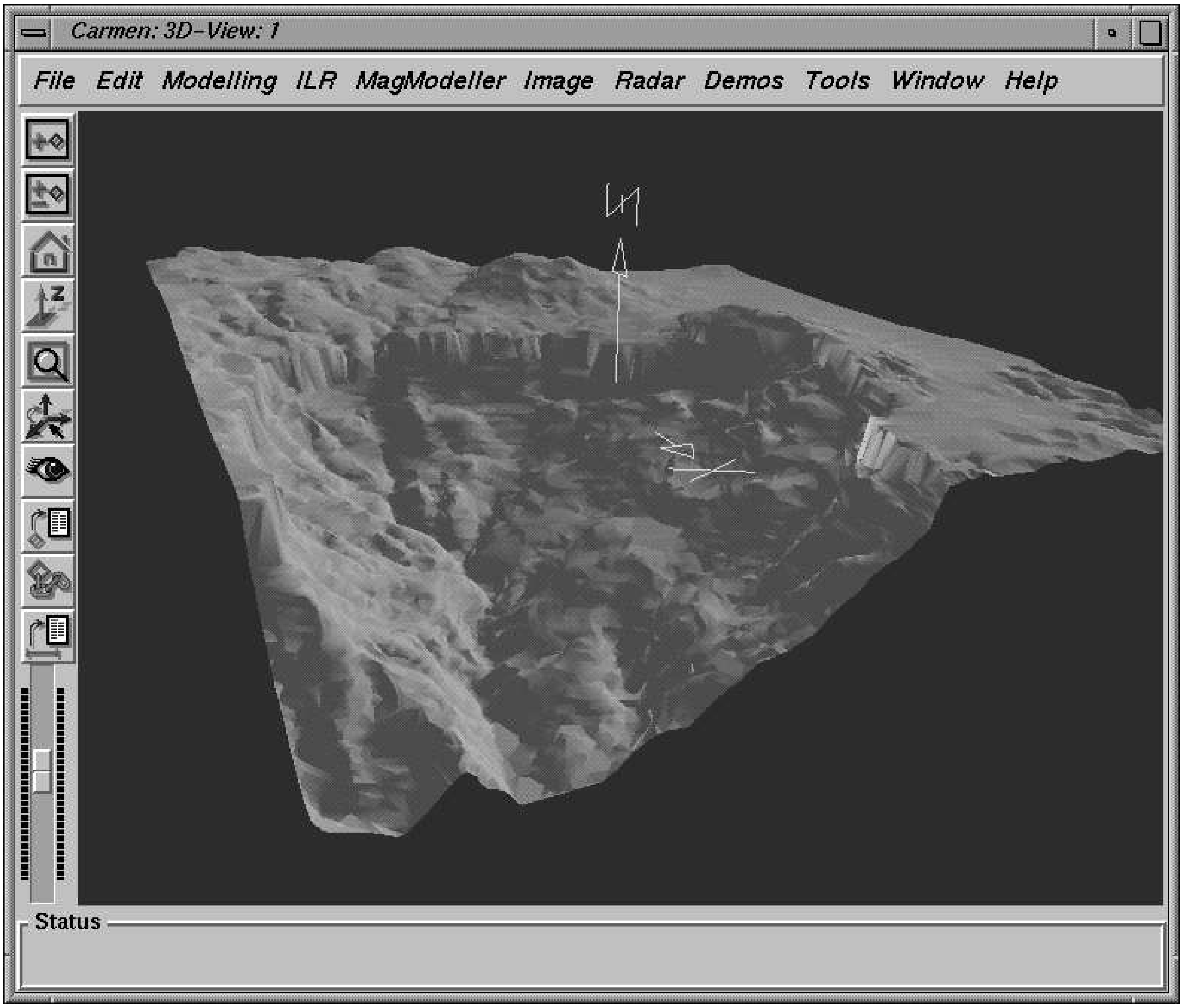

Figure 5.3: 3D visualisation of a basin using the Carmen Master Application.

5.2 A Carmen Web Server

Previous research has involved the implementation of an application to remotely access Carmen via the Internet. Through the provision of a Web server Gibbs [5] enabled the downloading of data from a Carmen Data Engine. This project builds upon that initial work which had a number of limitations:

- data could only be downloaded — there were no facilities provided to modify and upload new data to the Data Engine;
- no Carmen functionality was provided — the access allowed the browsing of the Data Engine with no other remote Carmen support;
- data was visualised using a third party software product because no facilities were provided by the application itself; and
- all of the interfaces used were static — any change to the Carmen end would require re-writing the interface code.

It is interesting to note that while Section 2.2 argues against the direct use of sockets, Gibbs application relied upon Java to Java sockets for communication. The protocol he designed was only concerned with "shipping" data between the client and server. As a result, the engineering overhead required to develop and implement the protocol was low compared to distributing the GUI. In terms of a trade-off between speed and engineering cost the use of sockets can be justified. Gibbs also investigated the possible use of CORBA but claimed that at the time of writing (late 1997) there was little support outside of commercial domains. This is no longer the case as pointed out in Section 2.4.

The Carmen Web Server developed by Gibbs represents important groundwork for this project. It demonstrates that remote access is possible via a platform independent application. The extension of this work is to provide access to all of Carmen's facilities by remotely implementing its GUI. The next chapter describes the first stage of this extension — the implementation of a Distributed GUI Server.

Chapter 6

Distributed GUI Server Implementation

Having developed the appropriate background and explored a suitable application for distributed GUI framework integration this chapter details the implementation of the Distributed GUI Server.

6.1 Java Native Interface (JNI)

Chapter 4 introduced the JNI. The JNI provides the mechanism by which the Java half of the Distributed GUI Server communicates with the C++ half and thereby Carmen itself.

The JNI essentially provides an interface pointer which accesses a lookup table. This arrangement is represented in Figure 6.1 [13]. To use the JNI shared libraries need to be created that are dynamically loaded by the JVM. Once these libraries have been loaded, calls can be made to and from C++(or C) as desired.

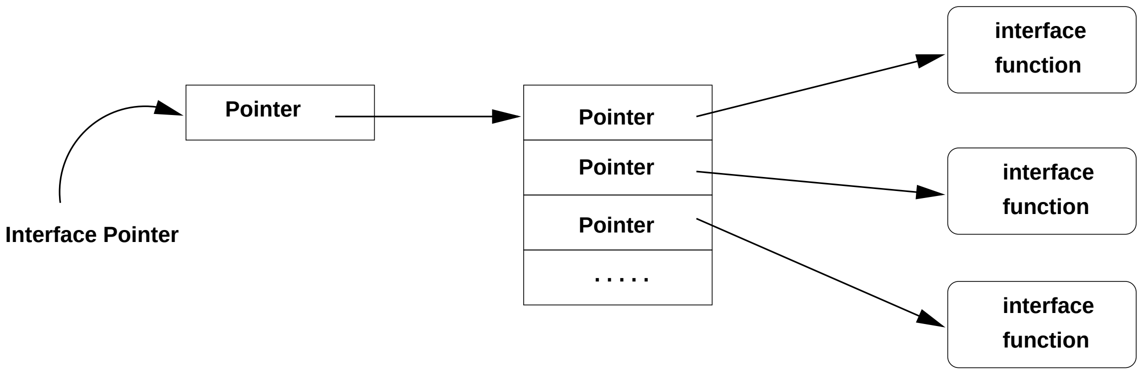

Figure 6.1: The Java Native Interface (JNI)

The JNI provides a number of support functions to allow C++ to look up class, field and method names as well as providing mechanisms to convert between string formats for example. Any C++ function that wishes to invoke a Java method requires the pointer. Care must be taken with the pointer because the JVM can also unload classes. If a C++ method copies the pointer then problems occur after that method returns because the class may be unloaded and the pointer is no longer valid. Section 6.3 returns to this point later in the chapter.

6.2 Architecture

While Carmen is a product still under development it could be considered in the same way as a legacy system in terms of web access. The integration of legacy systems is typically concerned with providing new interfaces for existing applications. This is typically to make them compatible with a new system or language. Another reason for incorporating a legacy system is to provide a new interface. This is typically the provision of a new font-end for some existing application. An example would be the provision of a GUI front-end for a text-based DBMS. The same scenario is effectively true for the system detailed here. This work is providing a new interface for Carmen. It looks and responds in the same way as the existing interface yet it is implemented using different technology [19].

One of the problems faced when integrating legacy systems is how to interface the new system to the old. For a re-usable front-end another problem may involve finding the appropriate level of abstraction. While Carmen's MCP offers an easy solution to the first problem the answer to the second is not quite so simple.

The Distributed GUI Server addresses the first problem by re-implementing all of the existing GUI server's functionality. By providing the same facilities as the original we avoid incompatibility problems and allow for the seamless integration of the new server into the Carmen environment. This functionality is provided by a combination of C++ classes, shared libraries and Java code.

The Distributed GUI Server works by effectively trapping widget requests that typically come via the HCLAF server. Rather than rendering the requests locally a remote proxy-widget is created by the server. This proxy-widget is passed the JNI interface pointer when it is constructed so it has the ability to invoke a corresponding widget in the client's JVM using RMI. This raises a potential problem. Once the method that instantiates the actual widget on the client returns, the proxy-widget cannot reference it. To solve this problem a widget hash-table is created by the client when it starts up. Whenever a proxy-widget invokes a method on the client it passes the value of its constructor in the remote method call. This constructor value is used as a key into the hash-table that makes the client widget available to the proxy.

One of the difficulties encountered when implementing this project was finding the right level of abstraction for the proxy widgets. If the level of abstraction was too high there would be no easy mechanism for utilising the required GUI functionality. On the other hand, if there was too little abstraction then the GUI server may be too closely tied to the toolkit used by Carmen to render its GUIs. This would create problems where Carmen's toolkit provided functionality that was not available in other toolkits. Carmen's widgets are implemented locally using the Qt toolkit briefly described in Section 3.3.2. While Qt's unique signal-slot technology makes Qt easy to use, it presents a problem when trying to map the same functionality to another toolkit. A level of abstraction needed to be found that provided access to necessary GUI functionality while masking Qt's unique features. The proxy-widget implementation used provides the appropriate level of abstraction.

6.3 Starting the Server

While Carmen's server based architecture provides for the integration of new servers there is a potential problem if two servers have the same name. If the "guiServer" is already connected to the MCP when another GUI server starts up then the MCP maps the new GUI server to a new name. Typically this involves prepending a number to the new server to produce a name like "1:guiServer". If we are trying to incorporate a rival GUI into the system then we have a problem. Existing Carmen servers and application code explicitly references the GUI server. It is undesirable to modify application code so that it can be used with the Distributed GUI Server. Ideally a way to integrate the new server so that its remote functionality is transparent to the application is required. Carmen's own GUI server is started by default for any applications that require its services. What we need is a way of taking the existing GUI server out of the equation. Figure 6.2 represents a way that this can be done.

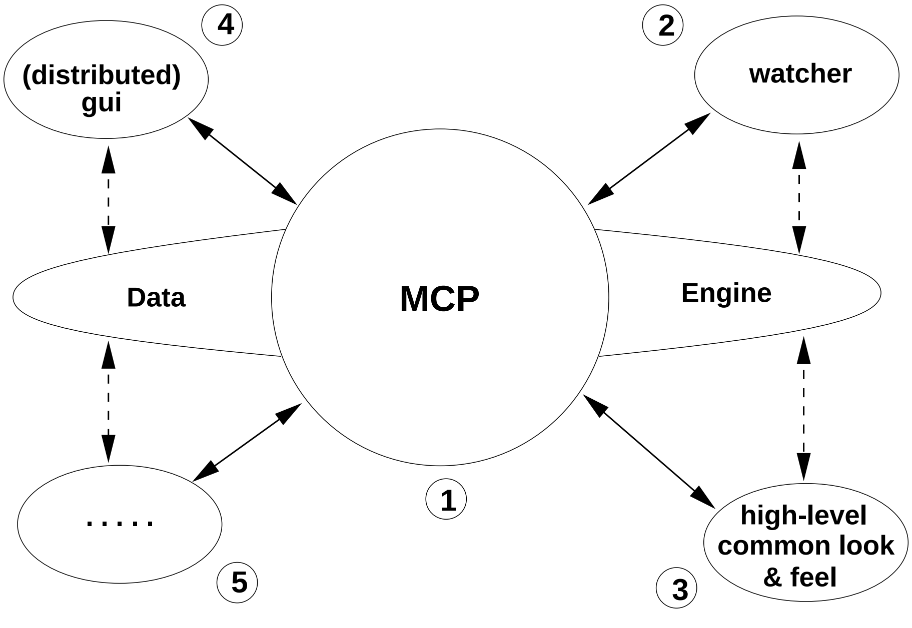

Figure 6.2: Starting the Distributed GUI Server.

Before a remote client can connect to the Distributed GUI Server the Java half of the server must already be running. When a client connects the Java half of the system can then launch the rest of Carmen. Rather than starting Carmen up in a generic way specific servers can be requested. Figure 6.2 represents an order for starting servers that will solve the GUI conflict problem. Step 1 is to start the MCP which in turn "activates" the DataEngine. Step 2 launches the Watcher server so that access to the DataEngine can be correctly monitored. The Distributed GUI Server relies upon instructions from the HCLAF server so this server is started at step 3. The base Carmen functionality is now running so the Distributed GUI Server can activate its C++ side via JNI and connect to the MCP. Here-in lies the deception. When the Distributed GUI Server connects to Carmen it authorises itself as Carmen's own GUI server. The MCP is quite happy to accept this because the Distributed GUI Server provides the same functionality as the original. Having safely started the GUI server without opposition any application can now be launched as represented by step 5. To Carmen's servers the disguised Distributed GUI Server looks and acts like the real thing — this provides the required integration transparency.

Having successfully integrated the Distributed GUI Server the question of client implementation can now be addressed.

Chapter 7

Distributed GUI Client Implementation

This chapter details the design of the client-side components of the GUI architecture. This represents the convergence Swing, AWT and RMI.

7.1 Architecture

The architecture of this project's client-side components fit into an emerging category of objects that are hard to classify as either a client or as a server. The Distributed GUI Client is so named to distinguish it from the Distributed GUI Server but both fulfill the roles of client and server. The GUI client is concerned with assembling widgets requested via the Distributed GUI Server into a usable, responsive GUI. An obvious pre-requisite to fulfill this task are appropriate widgets.

Section 3.3.2 introduced Java's Swing toolkit. Swing is a flexible GUI toolkit with a pluggable look and feel. It has suitable mappings for most of the Qt widgets used in the implementation Carmen's own GUIs. That fact that it is written in Java and provides the necessary functionality made it an obvious choice to implement the GUIs widgets.

However, before any widgets can be realised the GUI client first needs to connect to the Distributed GUI Server.

7.2 Remote Connection Window

Figure 7.1 is a screenshot of the Connection Manager or Remote Connection Window used for connecting to Carmen. As mentioned in the previous chapter the Distributed GUI Server must already be running on a remote host before a client can connect to it. When a user starts the Connection Manager it firstly creates a RMI object which is the GUI Framework Client. Once the Framework is ready the Connection Manager Window is displayed to the user.

The user enters the Internet address of the desired host into the host field at the top of the window. The default port for RMI is 1099. Other ports can be used and if it is known that the Distributed GUI server is listening on a different port then this value can be changed. The user then clicks on "Connnect" which starts the

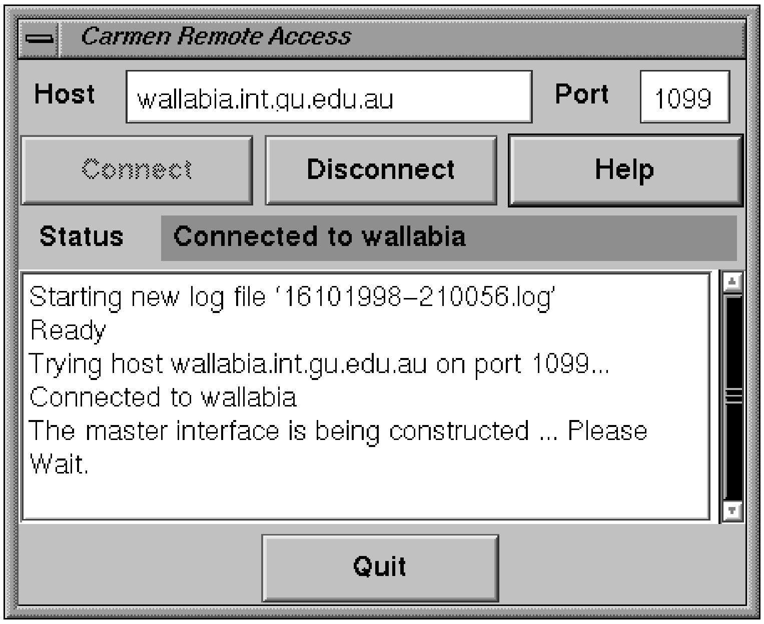

Figure 7.1: The Remote Connection window.

following sequence of events. Firstly, the Connection Manager tries to connect to the Distributed GUI Server which is itself an RMI server. The GUI client passes its host name to the Distributed GUI then connects to the GUI Framework Client. It also passes its host name on to the GUI client. This is used as a reference to implement callbacks. If the Distributed GUI Server can successfully connect back to the client then Carmen is launched in the manner prescribed by Section 6.3. The Connection Manager's status and log windows are then updated to reflect the state shown in Figure 7.1. This startup process is represented in Figure 7.2.

On the server-side the Distributed GUI Server starts Carmen and then launches the Master Application shown in Figure 5.3. Requests for the widgets to construct the Master Application are then forwarded to the client. On the University's LAN this process takes about 45 seconds from the time the user clicks the connect button until Figure 7.3 appears on the users screen. This Figure is an actual screenshot of the Master Application. It is constructed entirely from Swing widgets with a tweaked Motif look and feel. All of the tool-tips and menu's appear as they would on a machine running locally. This screenshot was taken with the GUI client running under IRIX. Figure 7.4 shows the same interface running on a Windows NT machine.

A number of basic interface design principles [24] are employed in the design of the Connection Manager Window. Firstly, the design has been kept simple and includes the provision for on-line help as shown in Figure 7.5. The "Status" and "Log" fields offer informative feedback to the user. This includes notification to the user of long delays as discussed in Section 3.2. The status messages and log descriptions also incorporate closure. For example changing the status message from "Connecting to 123.456.789.10" to "Connected to 123.456.789.10" indicates that the connection has

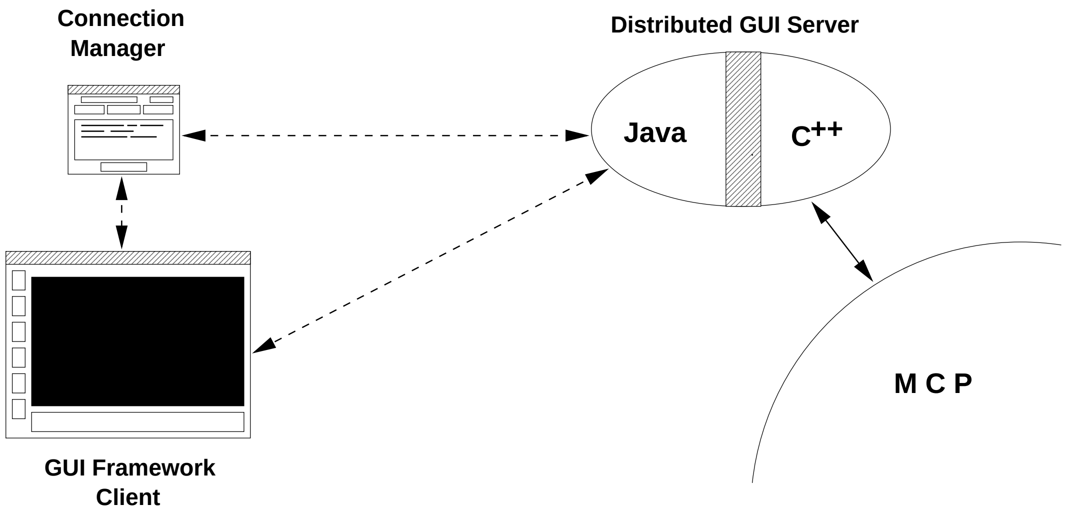

Figure 7.2: Connecting to Carmen.

been established. Buttons are enabled in a context sensitive manner. For example, the "Disconnect" button is only enabled after connection has been established.

The "Connect", "Disconnect" and "Help" buttons have obvious meanings — establishing a connection, shutting down a connection and providing additional information, respectively.

The "Status" field provides a brief indication of the state of the connection while the text area provides longer descriptions of error messages for example. This information is written to a file creating a session log. Finally, the "Quit" button shuts down the Server and Master Interface processes before closing the window. While the Framework Client is implemented using Swing, the Connection Manager and Help Windows are entirely AWT based. One reason for this design is so the Connection Manager looks more consistently like a native application. This enhances the distinction between the Master Application Window and the Connection Manager reinforcing their different purposes to users.

7.3 Distributed Callbacks

The GUI Framework effectively implements what Mowbray and Malveau term distributed callbacks [19]. Instead of using the Qt signal-slot mechanism Swing handles user interaction by defining "action listeners". These can be attached to widgets and in response to a particular event — for example a button click — they invoke a method. When an action listener is created the GUI framework "primes" it with the pointer value of the widgets corresponding proxy-widget in the Distributed GUI Server. As mentioned at the start of this chapter the Distributed GUI Server provides both client and server functionality. Part of that functionality is the provision of callback methods. When an action listener responds to an event it invokes a

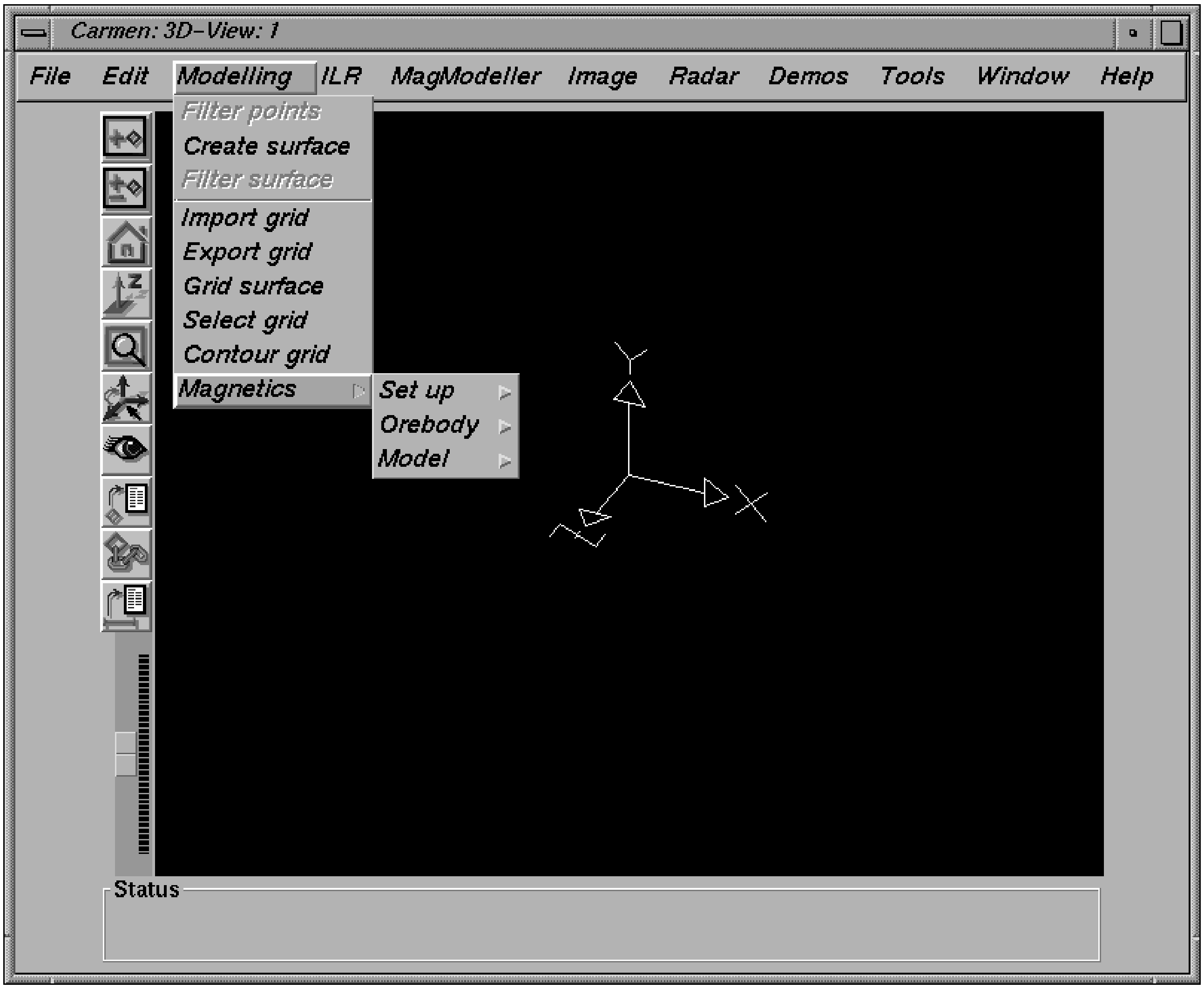

Figure 7.3: The Master Application GUI on the IRIX platform.

callback method in the Distributed GUI Server. The pointer value is cast back into a valid pointer which is then used to initiate a callback via the MCP. At this point the callback method invoked by the button returns. There is no additional delay because the callback does not need to wait for the completion wait for the "completion" of the call.

7.4 Other Implementation Details

The GUI Framework client was implemented using Java 1.1.6 which is the latest stable version of Java for IRIX 6.3 and Swing version 1.0.3. The current version of the Java classes and "gif" images required to implement the Master Application take up less than 200 kilobytes of disk space.

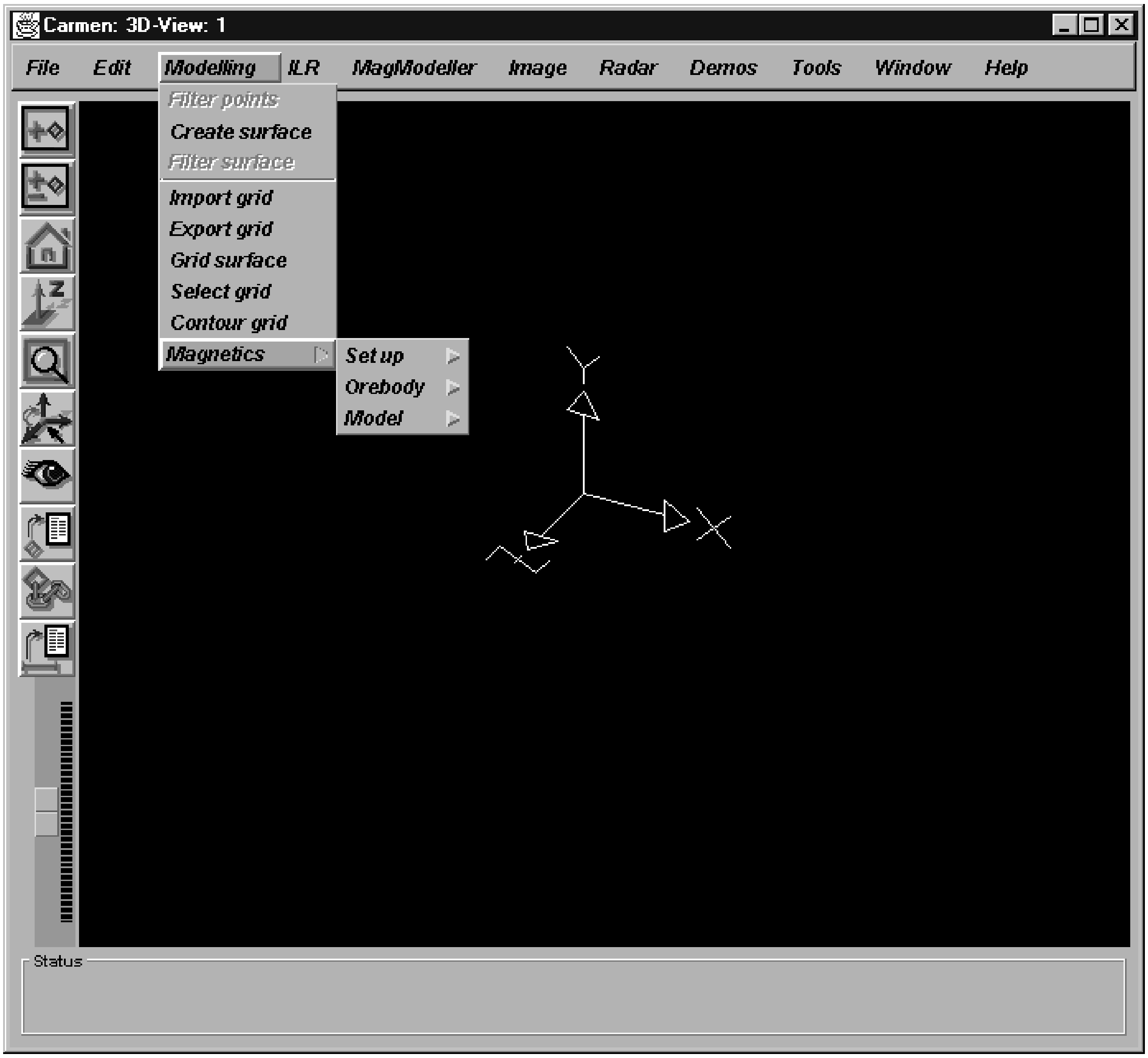

Figure 7.4: The Master Application GUI on the Windows platform.

The "gif" images are used to render some of Carmen's buttons. To avoid additional construction delay the decision was made to distribute these image files with the classes. This present s a problem because the Carmen Master Application asks for images from specific directories. The GUI Framework handles the requests for these items transparently, dynamically mapping them to the appropriate name on the local file-system. This process also abstracts over the fact that different platforms have different path separators for example. The design aim has been to make the system as transparent as possible. While this has been largely successful, it does introduce some unusual side-effects discussed in the final chapter.

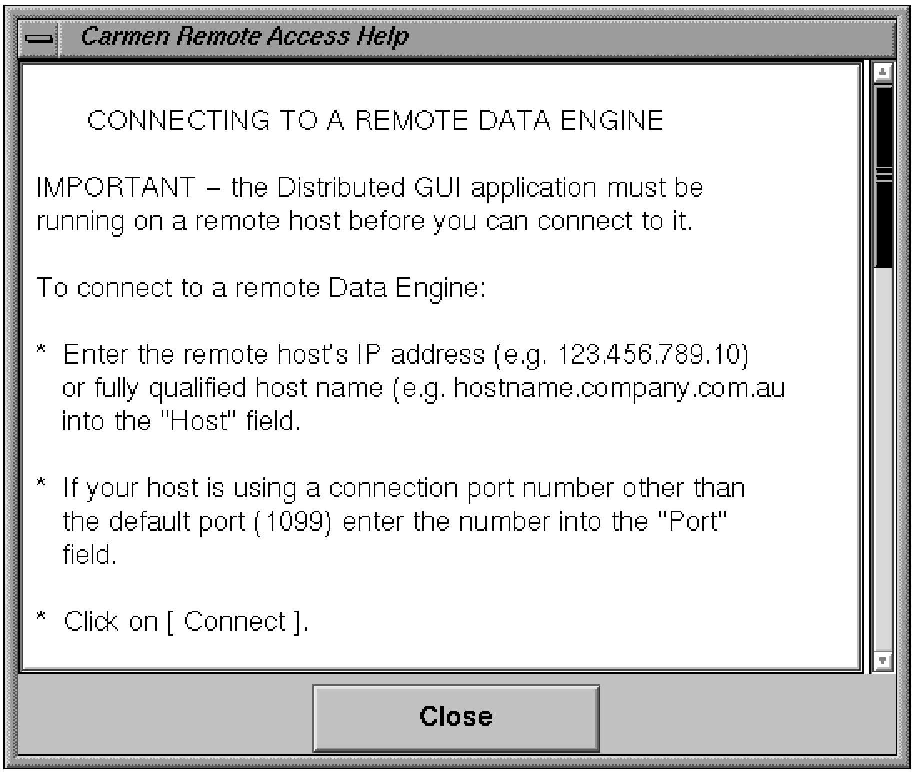

Figure 7.5: The Remote Connection Help window.

Chapter 8

Conclusions and Future Work

This final chapter presents the conclusions drawn from this work and suggests directions for further investigation. While this dissertation is focused on the remote implementation of Carmen's GUI it is clear from Section 5.1 that an essential part of this interface is the visualisation it provides. The closing sections of this dissertation discuss how this remote visualisation can be implemented. Finally Section 8.2.3 describes a possible scenario which could utilise the framework.

8.1 Conclusions

The first of the two aims from Section 1.1 was to evaluate some of the distributed technology options available to implement this project. Chapter 4 presented the experimental results obtained in a simple investigation of the performance of four different distributed implementation techniques.

While it is clear from these results that sockets based implementations provide the best performance in terms of speed there is a large engineering cost associated with their use. As described in Section 2.2.1 to provide any useful functionality a suitable application-level protocol would need to be designed and implemented. When weighed against the constraints imposed upon an Honours project, the time required to design and implement a suitably complex and robust protocol made the use of sockets infeasible. Where performance and shipping time need to be optimised, such as in the case of the Viewer server sockets are recommended.

The CORBA and RMI implementations have lower performance but they both provide high-level functionality. CORBA's ability to inter-operate between Java and C++ alleviates the need for explicit language conversion but its engineering costs were still considered too high for the distributed GUI. Although the use of RMI necessitated the introduction of JNI into the server, when weighed against CORBA's complexity and the associated learning-curve, the RMI/JNI combination still had a lower overall engineering cost for the distribution of highly structured, low bandwidth GUI information.

Chapters 6 and 7 described the implementation of the distributed GUI framework with RMI providing the underlying communication. This re-introduces the projects second aim and its hypothesis. While it is difficult to convey the operational style of a GUI with static screenshots the comparison of Figures 7.3 and 7.4 with Figure 5.3 does reflect the implementation's success in capturing Carmen's look and feel. The distributed GUI framework architecture of Figure 1.1 was realised within the imposed time constraints.

The other criteria by which the implementation is to be judged is its performance. While no timing figures are presented the Master Application GUI created by the framework does meet the response-time guidelines set out in Section 3.2. Users of the system are informed of the initial delay while the interface is being constructed. Once realised it conforms to Shneiderman's guidelines. On the basis of the architectures successful implementation and its operational performance the hypothesis embodied by this architecture is accepted.

8.2 Future Work

From the comparison of the screenshots it is obvious that there is a significant limitation to this work. Carmen's usefulness relies on its ability to render 3D visualisations of the objects in its spatial database. The focus of this work has been on the implementation of a framework concerned with the construction of GUIs at a widget level. A usable remote implementation of Carmen needs to provide some visualisation capability. Section 8.2.1 discusses how this could be incorporated into the existing architecture.

Another issue to be investigated is which general application features are required to make the use of a distributed GUI framework viable. Carmen's server-based OO architecture influenced the design of the Distributed GUI Server and allowed for its easy integration. Provided a new target application has a Motif look and feel then no modification is required to the client-side software. The Distributed GUI Server would however need to be modified for use with any alternative application. To enable remote GUI construction the applications requests for widgets need to be "trapped" before being realised locally. These instructions can then be forwarded to the client. For most applications this would require software modification and this is undesirable. Ideally, the integration of a distributed framework should be transparent to the application.

While the Distributed GUI Server has been transparently incorporated into Carmen the technique used does introduce some unusual side-effects. When a remote user selects a menu item a callback occurs. This callback forwards a number to Carmen that identifies which menu item was selected. An application can then respond to fulfill the menu request. The problem occurs when the user selects a menu item that invokes a call to the local operating system. For example, Carmen's Master Application has a menu item that opens a text editor for the user. If a remote user selects this menu item then the application opens the text editor on the host running Carmen — not on the the remote user's machine. When nothing seems to be happening in response to the users request they may reselect the item which will simply open another text editor on the host. Each text editor consumes system resources and a persistent user would be effectively executing a denial of service attack on Carmen. An alternative menu implementation or a technique to trap this type of behaviour needs to be investigated.

The current implementation does not take into account security considerations. Only the default security managers provided by RMI have been used. The security issues associated with distributed GUI systems need to be further investigated.

8.2.1 Visualisation

The incorporation of Carmen's 3D visualisation capabilities would require a significant amount of work. To parallel the functionality of the existing Viewer server a new Distributed Viewer Server would need to be integrated with Carmen. This revised architecture is represented by Figure 8.1. Note that no corresponding viewer client is explicitly represented in this new architecture. The existing Connection Manager could be modified to incorporate new visualisation functionality.

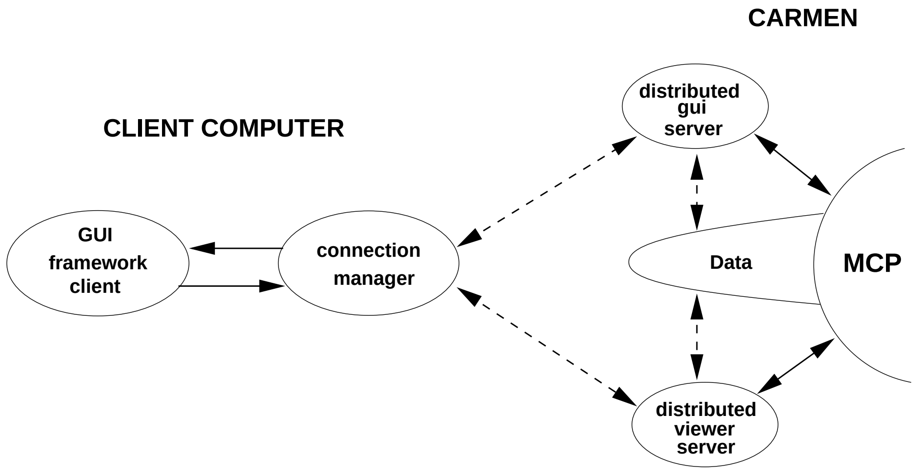

Figure 8.1: Incorporating a Distributed Viewer Server into the architecture.

The latency inherent in the distribution between Carmen and the remote client means that any real-time 3D rendering of spatial data needs to take place on the client. This implies that the data needs to be located on the client. A Distributed Viewer Server will not need to be concerned with providing visualisation. Instead it needs to handle the transfer of data to the client so that it can be rendered locally. This is effectively what Gibbs' work provided. His web server facilitated the downloading of data which could then be visualised on the client.

As pointed out in Section 5.2 Gibbs' work relied on third party software to provide visualisation so his work cannot simply be "embedded" to produce a viewer client. His implementation does however re-introduce an important implementation point. Sockets can be used to efficiently transfer data for visualisation. Depending upon what needs to be visualised large amounts of data may need to be shipped between the server and the client. To minimise delays to the user the time taken to download data must be minimised. A sockets-based implementation with a simple protocol would provide this functionality.

A final implementation consideration concerns communication between the GUI and Viewer servers. When a Carmen user manipulates a 3D visualisation communication occurs between the GUI and Viewer servers. This communication is routed via the MCP. For a remote user in this new architecture communication between the Distributed GUI client and the distributed Viewer client would still be sent via the MCP. If the implementation "trapped" these communications they could be kept intra-client and remove additional latency.

The remainder of this section examines the applicability of OpenGL and VRML to provide the required visualisation.

OpenGL

OpenGL is an open standard API (Applications Programming Interface) for developing two and three dimensional graphics applications [21]. Originally developed by Silicon Graphics it has become the dominant industry API in this area. It is platform and operating system independent which allows the application code to be ported easily between platforms. OpenGL applications can be split into a client and a server to support application distribution over a network. For example, the server could be a supercomputer running a simulation and the client a workstation displaying a visualisation of the results. The client and server can run on different platforms and there is an existing network protocol for sending and receiving OpenGL commands.

OpenGL bindings are available for both C++ and Java [1]. There are also a number of 3D graphics cards for PCs that provide hardware support for OpenGL. This makes OpenGL a likely implementation choice for any new visualisation services.

An alternative to using OpenGL directly will be available soon via the new Java 3D API [16]. Java 3D will provide some platform independence by relying on existing lower-level immediate-mode rendering APIs on target platforms. For example the IRIX implementation would use OpenGL to provide the rendering for the Java 3D methods. The new API has not yet been release but a Beta version is available for developers.

Virtual Reality Modeling Language (VRML)

VRML97 is the latest specification of the Virtual Reality Modeling Language (VRML) [9]. It enables the creation of complex 3D multimedia objects that can be viewed and traversed over the Internet using a VRML compliant web-browser. Authoring tools are available for many platforms including Java. It is designed to be both extensible and scalable allowing for the addition of new objects not explicitly defined in VRML for arbitrarily large 3D worlds. Some pre-defined object

[1] http://www.arcana.co.uk/products/magician/

manipulations are available and it has provision for the addition of simple effects such as fogs and cloudy skies. In terms of visualisation implementation VRML has a number of attractive features:

- it is an existing standard for viewing and manipulating 3D models;
- the standard was designed to allow the automatic translation of other 3D file formats into VRML files;
- it provides remote access to these models via the Internet; and
- the interface software is already implemented and widely available as a web-browser.

The disadvantage of VRML is that the worlds created with it are static. The data on which a model is based cannot be altered dynamically and the movement of objects within the world must be pre-defined. These object movements are little more than animations. Using VRML to provide visualisation would also require some way of embedding it into the viewer client.

8.2.2 Performance

A good distributed systems design limits the number of calls made to remote objects. While the performance of the current system is acceptable some obvious optimisations are possible.

In OO systems, an objects fields are typically accessed using "setters" and "getters". These are methods that set the value of a particular field or return the value of a the field. With a text-label widget, for example, we are likely to have a set-label method for updating the label with a new message and a get-label method which returns the label's existing message. Every invocation of these methods in the current system would invoke a remote call. If our design goal is to limit the number of remote calls then, instead of implementing both set-label and get-label remotely, we can use an alternative design. Obviously the system needs to be able to update labels so we must implement the set-label method remotely. However, to avoid the expense of a remote call to get the value of a label we could cache that information on the server-side. When we set the label a copy of the label value is stored locally as well as forwarding the appropriate message to the remote label. Any subsequent requests for get-label can then return the local value without invoking a remote call. This technique has been incorporated for some widgets in the current implementation but it use should be expanded. While it does offer a reduction in the number of calls to remote objects getters are typically called less often than setters.

8.2.3 Handling Multiple Clients

While the MCP and DataEngine have the ability to support multiple concurrent users the Distributed GUI server, and indeed Carmen's own GUI server, do not. Gibbs makes the observation that for a server to be of practical value it must be able to handle multiple concurrent clients. Carmen does not currently provide multi-user facilities. Gibbs implementation allowed a single server to support multiple users but his system only interacted with the DataEngine and the MCP. His web server was simply serving requests for data from the Data Engine. The GUI server, by contrast, interacts with both the HCLAF and Viewer servers to produce an interface containing a visualisation. To handle multiple clients the GUI, HCLAF and Viewer servers would all require modification.

An alternative approach would involve starting new HCLAF, Viewer and GUI servers for each client. This approach presents two problems, one is the use of computational resources. Each new client requires the services of at least three new servers.

Another problem is that the server names are unique and are referenced by each other. As described in Chapter 6 a second GUI server connecting to the MCP would be registered with the MCP as "1:guiServer". The existing code for all of these servers could be modified to provide some dynamic naming facility so that "1:guiServer", "1:HCLAF" and "1:ViewerServer" interact with each other rather than the "original servers". Alternatively the MCP could implement some form of dynamic name mapping. Requests for the HCLAF, GUI or Viewer servers could be mapped to those serving a particular client and application. As the MCP routes all requests it could also redirect responses to the appropriate place. The MCP already implements a certain degree of name mapping. This approach requires the extension of its existing functionality.

Example Application

Section 5.1 indicated that Carmen's client/server architecture allows the addition of new servers. One proposed application server is a flood simulator. The implementation of the remote GUI as a framework means that any new application added to Carmen is immediately accessible. Provided the flood simulator's interface uses standard Carmen widgets then no recompilation or additional changes are required to access it.

If the visualisation and multiple client issues could be addressed then Figure 8.2 represents a simple example involving the flood simulator. During a flood it is necessary to co-ordinate emergency services. The ability to survey the existing situation and make decisions about what will happen next is vital. In this scenario the relevant terrain information could be stored in a Carmen data engine. The weather bureau could access this terrain information using the GUI Framework Client and provide updates to the flood simulator. These updates could include such information as measured flood levels and recorded rainfall. It would then be possible for emergency services such as the Police and the State Emergency Service (SES) to access this same Data Engine to view the floods progress.

In addition to providing views of the current situation it would also be possible to run a simulation based upon the latest data from the weather bureau. A projection of areas that will be flooded in the next hour or the next day could help identify areas that need to be evacuated.

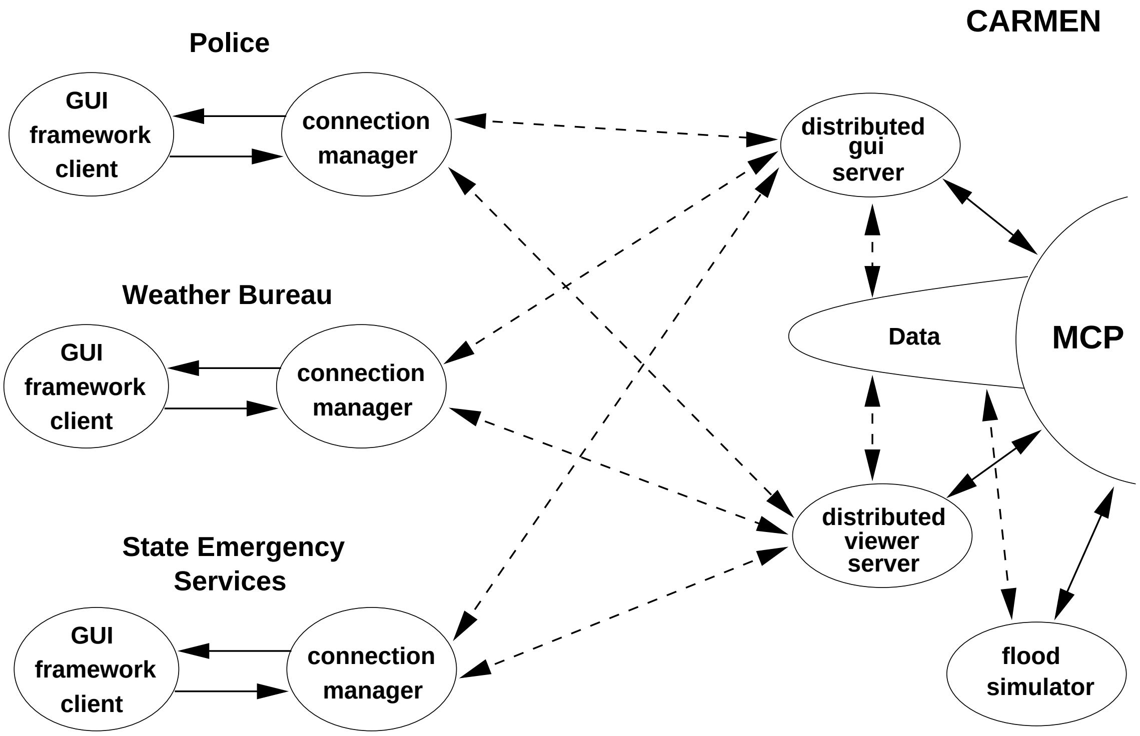

Figure 8.2: Example application.

The actual implementation of such a flood server is outside the scope of this project however it serves to illustrate a possible application for this research.

Bibliography

[1] A.E. Crane and S.W. Clyde. Extending patterns for GUI design. Research Paper, http://sys3.cs.usu.edu/students/AngelCrane/Final_merged.htm, 1997.

[2] I. Cyliax. Graphical user interfaces in RTOSs. *Circuit Cellular INK*, pages 46–53, May 1998.

[3] M.E. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] A. Gibbs. Platform-independent access to a remote spatial database management system. Honours Dissertation, University of Adelaide, 1997. http://www.int.gu.edu.au/kvo/reports.

[6] D. Griswold. The Java HotSpot virtual machine architecture. http://java.sun.com/products/hotspot/whitepaper.html, March 1998.

[7] The Open Group. Open Group desktop technologies - X windows system. http://www.opengroup.org/tech/desktop/x/, 1997.

[8] The Open Group. Network computing - a new business paradigm. http://www.opengroup.org/nc/, 1998.

[9] The VRML Consortium Incorporated. International standard iso/iec 14772-1:1997. http://www.vrml.org/Specifications/VRML97/, 1997.

[10] D.E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[11] D. Krieger and R.M. Adler. The emergence of distributed component platforms. *Computer*, pages 43–53, March 1998.

[12] R. McKay and W. Guthrie. Platform independent FAQ. http://www.zeta.org.au/~rosko/pigui.htm, 1997.

[13] Sun Microsystems. Java native interface specification. http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html, May 1997.

[14] Sun Microsystems. Java remote method invocation specification . http://java.sun.com:80/products/jdk/rmi/index.html, 1997.

[15] Sun Microsystems. Applets. http://www.java.sun.com/applets/index.html, 1998.

[16] Sun Microsystems. Java 3D API. http://java.sun.com/products/java-media/3D/index.html, October 1998.

[17] Sun Microsystems. Java foundation classes (JFC).

http://java.sun.com/products/jfc/, October 1998.

[18] Sun Microsystems. Write once, run anywhere: Why it matters. http://www.java.sun.com/features/1998/01/wora.html, 1998.

[19] T.J. Mowbray and R Malveau. *CORBA Design Patterns*. John Wiley and Sons, New York, 1997.

[20] B.A. Myers. UIMSs, toolkits, interface builders. to appear in: Handbook of UI Design, May 1996.

[21] OpenGL. Overview of OpenGL. http://www.opengl.org/About/About.html, 1997.

[22] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[23] D.C. Schmidt and M.E. Fayad. Lessons learned building reusable OO frameworks for distributed software. *Communications of the ACM*, 40(10):85–87, October 1997.

[24] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, Mass/ Menlo Park, Cal./New York, 1992.

[25] P. Sridharan. *Advanced Java Networking*. Prentice-Hall, New Jersey, 1997.

[26] A.S. Tanenbaum. *Modern Operating Systems*, pages 403–445. Prentice-Hall, New Jersey, 1992.

[27] Troll Tech. Qt reference documentation. http://www.troll.no/qt/, 1998.

[28] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogenous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[29] A. Vogel and K. Duddy. *Java Programming with CORBA*. John Wiley and Sons, New York, 1997.

[30] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, http://www.smli.com/techrep/1994/abstract-29.html, November 1994. SMLI TR-94-29.

[31] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, July 1989.