A Distributed Adaptive Window System

Duncan Paul Grisby

Queens' College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

March 1999

Abstract

In today's world, users of computer systems are far more mobile than ever before, and therefore wish to access their computing facilities from many different locations. At present, the most common solution to this is for people to carry at least some of their computing power around with them, in the form of notebook computers and personal digital assistants. Such devices are of course useful, but they inevitably provide a weaker operating environment than that provided by less mobile computing infrastructure.

An alternative solution is to provide mechanisms by which users can access their standard desktop environment from any networked computer. Although there are a number of existing systems which provide this functionality, they tend to be very sensitive to network latency, and thus can only realistically be used in the local area. Furthermore, they work by providing a homogeneous view of all possible systems, and so are unable to make use of advanced hardware and software facilities supported at some locations but not others.

This dissertation presents the design and implementation of a new architecture for creating highly distributed interactive software systems, based upon mobile, extensible CORBA objects. In particular, it details a novel distributed adaptive window system, named DAWS, which supports mobile users in a very flexible manner.

The argument of the dissertation is that existing systems fail to perform well when the user is mobile in a wide area, because their design parameters were only valid for a limited range of operating environments. By contrast, DAWS is constructed upon a framework which permits the window system to adapt, at run-time, to a wide range of different operating environments, in such a way that it can minimise the effects of latency, and make full use of whatever facilities are available.

In addition to the contribution of a new highly-flexible window system, the underlying framework demonstrates the flexibility of CORBA, and provides some useful techniques for other CORBA-based applications.

To Aviva

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes, and bibliography.

Acknowledgements

I would like to thank my supervisor Andy Hopper for always being helpful and approachable. I would also like to thank all of the staff at AT&T Laboratories Cambridge, especially Sai-Lai Lo, Quentin Stafford-Fraser, and David Evers for all their time, support and ideas.

Thanks too to all the members of first the OMRG at the Computer Laboratory, and then the LCE at the Engineering department. Special thanks to John Bates, and James 'Wez' Weatherall for reading and commenting, and to Margaret Levitt and Gilly Bowden for being infeasibly well organised. Thanks are also long overdue to Robin Walker for helping in all sorts of respects.

And finally, huge thanks to Aviva for putting up with me.

This work was supported by a grant from the EPSRC, and a CASE studentship funded by AT&T Laboratories Cambridge (formerly Olivetti and Oracle Research Ltd).

Contents

List of Figures

List of Tables

Glossary

API	Application Program Interface
BOA	Basic Object Adaptor
CORBA	Common Object Request Broker Architecture
DAWS	Distributed Adaptive Window System
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
ICA	Independent Computing Architecture
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
LAN	Local Area Network
MAN	Metropolitan Area Network
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
PDA	Personal Digital Assistant
POA	Portable Object Adaptor
RDP	Remote Desktop Protocol

RFB	Remote Frame Buffer
RPC	Remote Procedure Call
RRE	Rise and Run-length Encoding
TCP	Transmission Control Protocol
VNC	Virtual Network Computer
WAN	Wide Area Network

1
Introduction

This dissertation presents the design and implementation of an architecture for creating highly distributed interactive software systems. In particular, it details the design of a new window system, DAWS, which meets the requirements of today's and tomorrow's users far better than existing systems.

1.1 Motivation

In today's world, users of computer systems are far more mobile than ever before. Faster and cheaper transport, greater job flexibility, and the increasing influence of multinational companies all contribute to a working environment in which users wish to access their computing facilities from many different, widespread locations.

The most common solution to this at present is for people to carry at least some of their computing power around with them, in the form of notebook computers or personal digital assistants (PDAs). This is of course useful, but it does lead to new problems with keeping mobile information synchronised with information stored in the fixed computing infrastructure, and results in an obvious personal single point of failure. At present, and for the foreseeable future, there are resources, such as huge dynamic databases, which can only be accessed as part of a fixed infrastructure.

An alternative response to the problem is to provide mechanisms by which users can make use of the existing computing facilities available at their current location. In particular, it is useful for them to be able to access their standard desktop environment, including all of their running applications, wherever they may be. The increasing ubiquity of computer networks, all connected to the global Internet, and the trend towards an environment consisting of many relatively simple network-attached devices suggests that this will be a major mode of personal operation in future.

Building interactive systems which operate effectively in a heterogeneous environment in which resources, especially computing power, network capacity and low network latency, are plentiful in some locations but scarce in others is a great challenge.

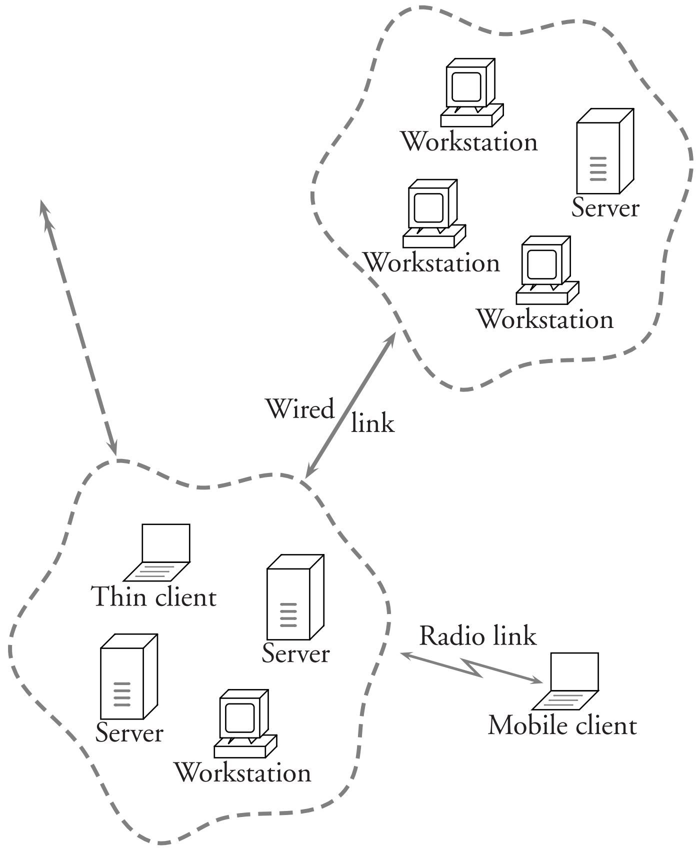


Figure 1.1: Islands of connectivity

1.2 'Islands' of connectivity

A key concept in this work is that of 'islands' of connectivity, as shown in figure 1.1. Each island is a group of machines in a local-area network (LAN). There is a wide range of different types of computer within an island: high-powered special-purpose servers, general-purpose workstations of varying power, thin-client network computers with little on-board processing power and storage, and possibly small special-purpose machines performing simple tasks. Moreover, these machines can support a wide range of input devices, from conventional mice and keyboards, to more exotic facilities such as sound and vision.

Machines within an island have low latency, high bandwidth, high availability connectivity to each other. From a networking point of view, it is essentially irrelevant where within an island a service is provided—access to the service is good from all locations within the island. Service placement can of course be optimised, but decisions are reasonably easy to make, and the scope for improvement is limited.

The situation is very different in the wide area: islands only have a small number of connections to other islands, and the links tend to be of higher latency, lower bandwidth, and lower availability than those within an island. There may be a small number of intermediate levels—a group of LANs could be formed into a metropolitan-area network (MAN), for example—and some islands will be better-connected than others. Placement of services between islands can affect the performance of a distributed system by many orders of magnitude.

Finally, an increasing number of machines are sporting radio-based network interfaces, allowing a user to roam freely while using network services. Current wireless networks tend to be either high bandwidth and short range, or low bandwidth and reasonably ubiquitous, such as cellular telephone systems. Some machines may have both types of interface, and be able to switch between using their high capacity interface when near an island, and their low capacity one when in the 'sea' between islands. Because of their mobile nature, wireless machines are always likely to be more limited in their processing, storage, and networking abilities than machines which are part of fixed infrastructure.

An interactive distributed system should provide effective support to users as they travel within and between these different environments. Users roam freely within an island, and less regularly move between islands. They may have mobile computers with wireless network interfaces, but quite probably prefer to use the fixed infrastructure when they are within an island. Most importantly, they expect the state of their interactive sessions to be preserved and available wherever they are.

It may be tempting to assume that most of these problems would be avoided by simply improving the networking infrastructure between islands. That would, indeed, ease the situation. However, the quantity of network traffic will inevitably grow to fill the available bandwidth, and the way systems are built will mean that there is usually more network capacity available between machines physically close together than those far apart. Aside from this, the speed of light imposes an ultimate limit on how low latency may become.

1.3 Contribution

Users wish to be able to access their running desktop environments as they move from location to location. A natural way to provide this functionality is for the window system that the user runs to be able to dynamically change its end-point—the combination of display and input devices—to one that is near the user.

The argument presented in this dissertation is that conventional window systems, and software systems in general, are unable to extend to support users who are highly mobile in a wide area. This is understandable because the design parameters for these systems were valid for only a limited range of operating environments.

Today's highly mobile users move between a much wider range of environments than was ever envisaged by the designers of current systems. It is the thesis of this dissertation that:

- Future systems must be designed to adapt to a much wider range of design parameters than current systems.
- A powerful approach to this is to use a component-based framework in which components may vary their locations and implementations in a dynamic fashion.

To validate the thesis, this dissertation presents a new distributed adaptive window system, DAWS, which is highly flexible and extensible. The system is based upon a framework of CORBA objects which are able to migrate between locations.

1.3.1 Alternative solutions

Of course, making the window system aware of user mobility is not the only possible way of permitting the user's desktop to follow them. The other main approach is to make the *applications* that the user runs aware of mobility[BHB96, BC96, Pop96]. In this scheme, all applications must have explicit support for the migration of at least some of their functionality.

This sort of design has a number of advantages, most notably that applications can achieve better performance by migrating, and that they may be able to intelligently adjust their behaviour based upon the user's circumstances. Unfortunately, a system which only employs application-level migration has the drawback that it relies on the applications being well-behaved with respect to following the user: if an application fails to follow the user, there is nothing the system can do. Also, maintaining the desktop's visual appearance and arrangement across migrations is more challenging without support from the window system.

DAWS advocates a combination of the two approaches, in which the window system provides the basic user-following capabilities, but individual applications can also migrate if it is beneficial for them to do so. The framework upon which DAWS is built is a general-purpose one, and may be used by migration-aware applications.

1.4 Terminology

It is important to understand what is meant by the term *window system* in this dissertation. The desktop environment experienced by the user is formed from two elements, the window system, and the graphical user interface (GUI) system built on top of it. In many systems, such as Microsoft Windows and the Apple Macintosh, the two elements are intrinsically linked (and are also linked to the operating system); other systems, like the X window system, do make a distinction.

It is the job of the window system to perform low-level functions, such as maintaining overlapping *layers* of pixels, and *clipping* items drawn into the layers. It must also handle input devices, such as keyboards and mice. There is not, however, a clearly defined limit as to which functions are part of the window system—some window systems, for example, may provide complex line drawing commands, while others leave such functionality to applications or GUI toolkits.

The GUI system is responsible both for producing interactive components of the environment, and for generating its graphical appearance. So, for example, although the window system provides the abstraction of a clipped layer, it is the GUI system which produces the imagery and interactive elements which form the user's idea of a 'window'.

The focus of this dissertation is on a window system upon which many different GUIs could be built. Some parts of a rudimentary GUI are also described, both to illustrate how a full GUI system would be constructed, and to permit evaluation of the system's performance.

The other element of terminology relevant to this dissertation is the classification of different systems as 'adaptive' or 'adaptable'. 'Adaptive' means that the system in question automatically adapts to its circumstances in some way. 'Adaptable' means purely that the system *can* be adapted, either automatically, or with human intervention.

1.5 Outline

The remainder of this dissertation has the following layout:

- Chapter 2 describes the architecture of the most common, and some less common, window systems currently available, and some extensions to those window systems which support user mobility. It explains why these systems are unsuitable for supporting widely-mobile users.

- Based upon the shortcomings of the systems described in Chapter 2, Chapter 3 summarises the requirements of a new distributed window system, and introduces the architecture which has been designed to meet those requirements. The architecture consists of a general-purpose low-level framework, upon which the window system is built; the remainder of the chapter is devoted to a description of the low-level framework.

- Chapter 4 details the design of the DAWS window system itself, which is built upon the framework described in Chapter 3.

- Rather than make all parts of the system aware of the mobility of components, the architecture makes use of a general object migration scheme. Chapter 5 explores various possible designs for the scheme, and details the approach and implementation selected.

- Chapter 6 presents some of the implementation issues of DAWS, and also gives the design and implementation of a number of user interface elements.

- Chapter 7 contains performance measurements and an evaluation of DAWS. It also presents some hypothetical situations in which the architecture's highly adaptive nature would be beneficial.

- Finally, Chapter 8 contains a summary of the main arguments of the dissertation and some suggestions for further work.

2 Background

This chapter describes and evaluates past and present windowing systems, at a reasonably low level, and explains why they are not suitable for supporting highly-mobile users. Research in the field of mobile objects and agents which has influenced the design of DAWS is discussed in later chapters, in conjunction with the design itself.

It is interesting to note that only one of the window systems in common use, the X window system, has any notion of network distribution. This is largely due to the different heritages of the systems—X was developed for high-powered networked Unix workstations, while most other systems started in an environment of non-networked desktop personal computers. Consequently, this chapter concentrates mainly on X and other, less commonplace, network-aware window systems.

This chapter also describes two approaches to adding limited support for user mobility to existing window systems. Both schemes split the window system functionality in a rigid manner, and thus make assumptions about the environment in which they will be used: the systems perform well in the local area, but badly in the wide area.

The chapter ends with the conclusion that, although backwards compatibility is generally useful, it must be sacrificed so that more adaptable window systems may be constructed.

2.1 Early window systems

The idea of a user interface consisting of direct-manipulation graphics controlled by a mouse was first demonstrated by Douglas C. Engelbart and his group in the late 1960s at the Stanford Research Institute[EEB67]. Researchers at Xerox's Palo Alto Research Center (PARC) built upon these ideas to produce the *Alto* and *Star* computers[SA88]. Although the Alto and Star both had built-in Ethernet networking[MB76] capabilities, their window systems had no distribution support.

The Star was launched as a product in 1981, at around the time that IBM released their first Personal Computer. Much to IBM's surprise, the PC rapidly captured the new market for desktop computers, and Xerox's innovative machines did not become popular.

After a false-start with the *Lisa*, Apple was the first company to bring a window system to the desktop of the average computer user, with the launch of the *Macintosh* in 1984. The Mac's window system is very tightly coupled to the rest of the system, leaving little scope for distribution[1].

2.2 Unix window systems

In the early days of Unix window systems, two architectures were explored: *kernel-based* and *server-based* systems[Fri87]. Kernel-based systems relied on extending the Unix kernel to support the operations required by a window system. Applications using the window system linked with libraries of graphics operations, allowing them to create windows and draw into them. Sun's *SunWindows*, distributed with their first workstations, followed this model.

Server-based systems run the window system within a single user-space process; applications communicate with the server through some inter-process communication mechanism. This avoids adding the complexity of a window system to the operating system kernel, and thus improves system stability. It also means that applications require far less library support for drawing operations, since such operations are now performed by the server. Most Unix window system designers quickly settled on this model.

2.2.1 The X window system

The X window system[SG92] was developed at the Massachusetts Institute of Technology (MIT) in the late 1980s, at about the same time NeWS (see section 2.2.2) was being developed at Sun. X has become the dominant window system for Unix-based computers.

There are three main elements to the base X system, as shown in figure 2.1: the server, the client libraries, and the underlying X protocol they use to communicate. The user's endpoint, consisting of display, keyboard and pointing device, forms the *server*, which runs as a user-space process in Unix; the applications are *clients* of the server, and may run on any machine[2].

Several components are built on top of the base system to form a GUI: each application has an *application interface* which presents the controls and visible appearance of that application; the *window manager* is a special X client which controls the size

[1] The VNC system described in section 2.5.2 can add limited support for user mobility to the Mac.

[2] This reverses the normal usage of the terms 'client' and 'server', where the user runs local clients to a central server.

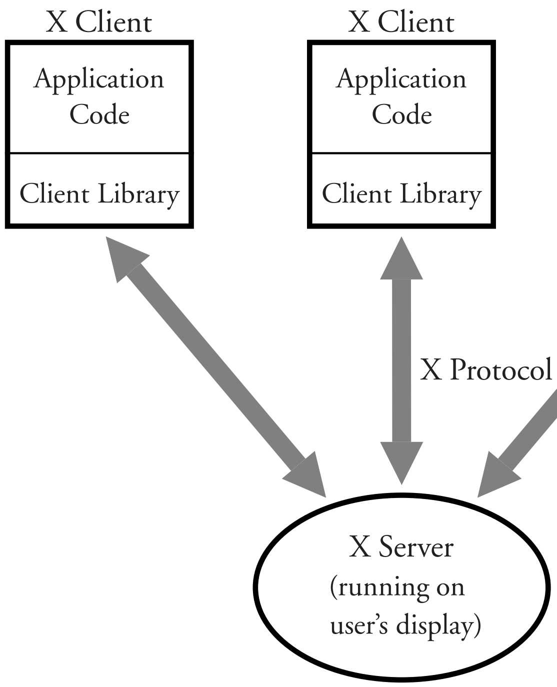


Figure 2.1: Architecture of the X window system

and placement of applications' windows, and maintains elements such as window title bars; the *input manager* controls which applications receive user input.

Design

The most important aspect of X is that it is a network-transparent system—X applications do not need to know whether or not they are running on the same machine as the X server which is providing the display. There were a number of other factors which influenced X's design[SG86]. MIT's environment at the time consisted of a number of different hardware platforms, with different display capabilities, so X was designed to be inherently cross-platform. Clients of the window system (i.e. applications) are insulated from platform differences by the X protocol.

X was designed to be flexible in a number of ways. Firstly, unlike some other window systems, such as NeWS, applications are able to present exactly the imagery they require, on a pixel-by-pixel basis, to the X server. This means that applications can make intelligent decisions about how best to present their GUIs. Secondly, the window system does not impose any particular user interface, either in terms of behaviour or appearance, leaving such concerns up to individual applications and window managers. Finally, the X protocol was designed to be extensible, so that new facilities could be added to servers.

Functionality split

The split in functionality between the X server and X client is determined by the X protocol[XPr90]. The split is rigid—it is not possible to configure the system to perform more or less processing at the server, even at compile time.

The protocol can run over any reliable bi-directional stream, usually TCP/IP, or Berkeley Unix domain sockets (for clients on the same machine as the server). Unlike earlier window system protocols[LN84, Now85], the X protocol is asynchronous, so as to maximise throughput by avoiding round-trip latency.

The protocol defines 120 types of *request* message which may be sent by clients. Most of these requests are serviced silently by the server, but some trigger a *reply* message. Since most requests do not have replies, many client requests can be in transit over the network at the same time, meaning that network bandwidth, rather than latency, provides the upper bound on rendering performance. Clients do not have to poll the server for user input either: the server asynchronously sends *event* messages for input in which the client has expressed an interest.

The protocol is such that the server is responsible for maintaining a hierarchy of rectangular windows, and also text fonts, mouse cursors, and off-screen images. It provides complex drawing primitives including straight and curved line drawing with flexible line widths and styles. It also provides shapes filled with a solid colour or a tiled pattern, pixel map drawing, and text rendering. All drawing requests are clipped to the boundary of a window[3].

Windows within X are very light-weight entities, and may be nested within each other. It is common for each user interface *widget* (buttons, scroll bars, etc.) to reside within its own window. Every window may have its own *mouse cursor*; the server automatically alters the mouse pointer image as the mouse moves between windows.

The server is not required to maintain the contents of obscured portions of windows, although it may if it wishes, based on hints set by its clients. If part of a window becomes visible, and the server is not able to re-draw it itself, an update event is sent to the client. Clients are not able to find out which areas of their windows are obscured in all circumstances. The one exception to this refresh mechanism is that transient windows such as pop-up menus are able to explicitly save the screen contents they are about to obscure. So that updates in other windows do not confuse this, the client *grabs* the server, preventing other clients from drawing until the menu is removed. The grab lasts until the client releases it, or the client exits.

Finally, the server is responsible for receiving user input, and sending input events to the correct clients. The server is expected to have a keyboard and a mouse; the two types of input are intrinsically linked since keyboard events contain the mouse state, and mouse events contain the state of the keyboard modifier keys (shift, control, etc.). Individual clients are able to temporarily grab one or both types of input to prevent

[3] Unless *draw-through* mode is used.

other clients seeing events. This is used during transient activities such as dragging icons, and to prevent clients snooping as the user types passwords. As with output grabbing, the grab lasts until the client releases it, or the client exits.

All other GUI functionality must be provided by clients. In particular, no interactive elements are provided by the server—all parts of the GUI which change their imagery in some way due to input activity are controlled by the clients.

Although it would be possible to write client applications which used the X protocol directly, it is normal to make use of an X library which provides a higher level interface. There are several levels of library, ranging from low-level calls which basically resolve to direct protocol requests, to high-level *toolkits*. Clients obtain GUI elements such as buttons and scroll-bars through *widget sets*. There are a number of widget sets in common use, such as *Athena* and *Motif*. Each widget set is a programming language library, with its own programming interface. It is not usually a trivial operation to change the widget set used by an application.

Shortcomings

X is generally quite successful and flexible, but is does have a number of general shortcomings[GM90, Hop94], and some specific problems which prevent it from being extended to support users who are mobile in a wide area.

General problems include the fact that it does not completely succeed in insulating clients from the hardware details of the display, especially in its support of different colour models. There are stability problems too—if a client grabs the server's output or input and then hangs, it effectively hangs the whole server, for example. When this happens, the user's only solution is to attempt to kill the client without using the window system.

At a higher level, the fact that clients (or client libraries) are responsible for the appearance and behaviour of the GUI, and there is no standard interface to this functionality, means that different applications tend to have radically different user interface styles. There is usually nothing that the user can do to force all of their applications to have a consistent 'look and feel'[Get98].

Despite the fact that the X protocol was designed to be extensible, X is, in practice, extremely rigid in its functionality. The only way to implement new window system services to make use of new types of hardware, or new interface ideas, is to compile the services into the server. This has had the result that a small number of extensions, such as the Shape and XInput extensions, have essentially become part of the X standard, and are supported by all X servers. No other extensions are in wide use.

Conversely, it is also very difficult to create a more limited X server. Research at Xerox PARC[KDF+93] attempted to adapt an X server to run on their MTab wireless displays by running a partial server on the MTab and the rest of it elsewhere. They concluded that X simply could not adapt to a system so different from the systems it was originally designed for.

The most obvious barrier to supporting mobile users with X is that the protocol has no means to instruct a client to disconnect from its current server and connect to a different one. It would, however, be reasonably simple to extend the protocol to this end, although modifying client applications may be quite challenging. The alternative is to use an X protocol proxy which can re-direct the end-point without the applications' involvement. The Teleporting system described in section 2.5.1 takes this approach.

A much more significant problem with supporting mobile users is the fact that X is very rigid with its network requirements. It was designed to operate in an environment where the round-trip time between client and server is between 5 and 50 milliseconds[SG86], and where there is plentiful bandwidth. It becomes very difficult to use when the latency becomes much larger than 50ms, or the bandwidth is low. A large number of systems[Cor92, FK93, Pac94, Dan94, Dan95] attempt to improve the situation by compressing the X protocol stream with varying degrees of effectiveness. These can be very effective over low bandwidth links, but add even more latency to the system, so are a partial solution at best.

If interactive parts of the GUI could be moved into the X server, it would be far less dependent on latency.

2.2.2 NeWS

NeWS[GRA89], the Network/extensible Window System, was developed by Sun as a replacement for the kernel-based SunWindows system[4]. It was well received when it was first released[RSD+87, LCW+88], but it ultimately lost out to X.

Like X, NeWS consists of a large server process in charge of the display and input devices, to which client applications connect. As with X, clients can be remote from the server. The major difference is that NeWS does not have a rigidly-designed protocol for communication between the clients and the server: applications communicate with the NeWS server in a far more ad-hoc manner.

The key design feature of NeWS is its extensibility[5]. It aims to provide a framework which supports a wide range of interface complexities—both for user interfaces and application interfaces. Applications wishing to perform advanced tasks may use a powerful and complex interface; those only performing simple tasks may use a simple interface.

The NeWS server achieves its flexibility and extensibility through the use of an extended version of Adobe's PostScript page description language[Ado90]. PostScript is a stack-based postfix language similar to FORTH with a large number of imaging primitives. NeWS adds a number of new primitives to PostScript, providing support for

[4]NeWS was originally known as SunDEW[Gos86], the Distributed Extensible Window system.

[5]It is therefore rather odd that the 'e' is the only letter not capitalised in 'NeWS'.

multiple light-weight processes, interactive events, and object-orientation with single inheritance. For each NeWS client, there is a PostScript process in the server which manages byte-stream communications with that client. Applications may create more processes within the server if they wish.

PostScript has a very powerful non-integer-based imaging model which can draw and clip to complex paths, using a system of *stencils* and *paint*. PostScript is completely device independent, and automatically adapts its output for the device to which it is rendering. So, for example, complex imagery can sometimes appear very degraded on low-resolution displays, and colours are automatically converted to grey or dither patterns when colour support is not available. The use of PostScript therefore makes high-level rendering, for activities such as WYSIWYG editing, very simple, but makes pixel-level drawing activities very difficult.

All input handling within NeWS is also performed with PostScript. When the user performs some input, an *event* is sent to a particular PostScript event handler, based upon the type of the event, and the currently active window[6]. Since the event handlers are written in PostScript, they may perform complex operations within the server, rather than relying on remote clients. For example, the 'rubber-band' displayed when a window is resized could be implemented entirely in PostScript, rather than by sending repeated messages over the network. In this way, NeWS is far less dependent on network latency than X.

This architecture means that all applications must have two elements: one written in PostScript running on the NeWS server, and the other written in any language communicating with the PostScript portion via a byte-stream. Much of the code running on the server is actually library code, rather than application-specific code. User interface elements such as window imagery and menus are all usually implemented in the server.

Like X, NeWS has no support for disconnecting clients from one server and reconnecting them to another, but, again, it would be possible to add such functionality with either protocol additions or proxies. NeWS has a significant advantage over X in its dependence on latency, since much more of the GUI's interactive behaviour can be implemented on the server.

There would, however, be a number of significant disadvantages in resurrecting NeWS in a form which could support mobile users. Firstly, although it is possible, and indeed encouraged, to place interactive components in the server, all of these components must be written in PostScript, which is a rather awkward language, especially when the NeWS additions are taken into account. Furthermore, NeWS has no support for running PostScript code anywhere other than on the server. This means that to construct a flexible system in which components could reside in either the client or the server, the components would have to be implemented twice—once in PostScript,

[6]Termed a *canvas* in NeWS.

and once in some other language.

NeWS also had stability problems, since it lacked proper scheduling and protection between PostScript processes. A modern re-implementation would be able to overcome these difficulties. Finally, NeWS requires a powerful server—more powerful than X, even—so it would not be feasible to adapt it to thin, simple display devices.

To some extent, much of the promise of NeWS has been reincarnated in the form of Java. Although not positioned as a window system, Java does permit applications (written using any language) to export portions of their GUI code (written in Java) to the user's end-point machine. Java solves some of the difficulties that NeWS encountered due to its use of PostScript, but many of the fundamental difficulties remain. In particular, the system can not extend to very simple end-points, and there is no scope for making use of special-purpose, esoteric devices without breaking outside the Java 'sand-box'.

2.3 Microsoft Windows

No discussion of window systems would be complete without mentioning Microsoft Windows, the current dominant personal computer operating system. Windows is actually several different operating systems, but all provide the same sort of window system facilities to applications, with roughly equivalent APIs.

The window system and GUI in all flavours of Windows are much more tightly linked to the operating system than they are in X or NeWS. It is not possible to modify the behaviour of the GUI, or the window management policy, without patches to the operating system. Of course, this has the advantage that the look-and-feel is forced to be reasonably consistent between applications, but only at the expense of options for the user.

Windows does provide a reasonably flexible API, allowing applications to perform rendering operations at both a pixel level (avoiding the problems of NeWS), and in a standard device independent manner (avoiding X's lack of standardisation in this respect). Device-independent output extends to printing support.

From here onwards, this section discusses Windows NT, as it is the most modern and flexible of the Windows operating systems, and it is Microsoft's intention that the NT architecture will eventually replace all of the other Windows desktop operating systems[7].

Figure 2.2 is a diagram of Windows NT's original architecture (as described by Custer[Cus93]). This shows that the window system runs as part of a user-level process which constitutes the Win32 sub-system. Other processes which wish to use the window system (and the other system services in Win32 sub-system) communicate with it

[7] Less capable devices will run Windows CE.

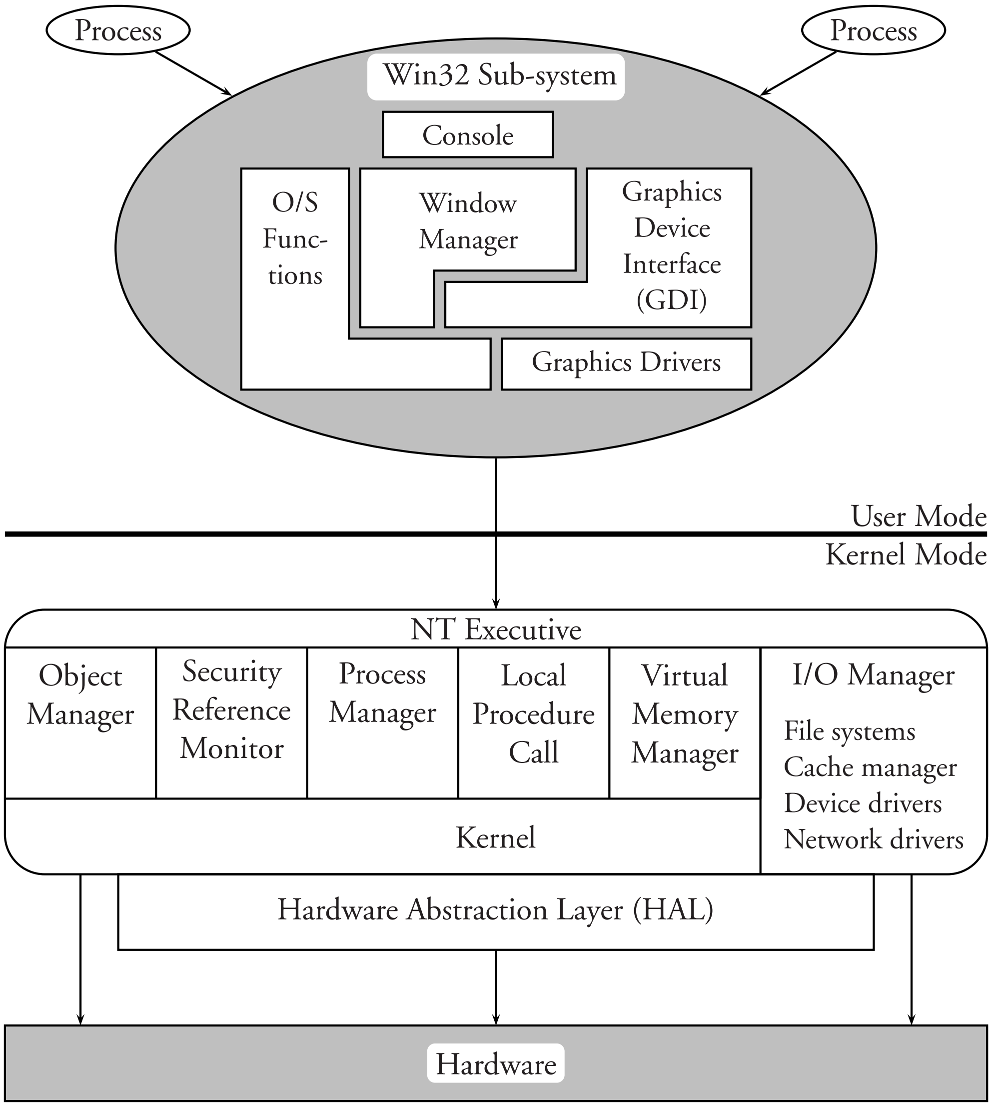


Figure 2.2: Architecture of Windows NT

via a *local procedure call* (LPC) mechanism. LPC is similar to an RPC (remote procedure call) mechanism, but optimised and restricted to communication within a single machine.

In the context of adding support for mobile users, this design is extremely unfortunate. Firstly, if a full RPC mechanism had been used for communication between clients and the window system, remote applications would automatically have been supported. Secondly, since the window system code is batched together with the rest of the Win32 sub-system, it requires significantly more work to modify the window system's behaviour. If the window system had been placed in its own NT sub-system, it would have been reasonably simple to replace the entire sub-system with a more flexible one.

To compound the problem, with the release of NT version 4, almost the entire window system was moved out of the Win32 sub-system and into the kernel-mode NT Executive. Any changes to the way the window system works now have to be made to privileged code.

ICA and NT Terminal Server

Support for remote access to Windows desktops by mobile users was added to NT version 3.5 by the *WinFrame* product from Citrix. Citrix achieved this feat by licensing the NT source code, and modifying the display system to output their ICA (Independent Computing Architecture) protocol, rather than rendering to the screen[8]. ICA is similar in complexity to the X protocol, and is thus dependent on latency to a similar extent. ICA *does* permit users to disconnect from their sessions and re-connect later from a different location.

With the release of NT 4, Microsoft licensed Citrix's multi-user technology back from them, and produced a special version of NT, named *Terminal Server*[Mic98b]. Terminal Server permits sessions to export their desktops using a protocol imaginatively named RDP, Remote Desktop Protocol. RDP is a much lower-level protocol than ICA, based upon the International Telecommunications Union's (ITU) T.120 standard[Dat97]. It basically represents a compressed form of the pixels to be rendered on-screen, and allows input events to be returned[9]. Terminal Server is extensible, and allows different protocols to be added—Citrix's *MetaFrame* product adds support for ICA.

Terminal Server, like WinFrame, supports mobile users by maintaining sessions as the user moves between different end-points. Both products do a good job of supporting users who move within an island of connectivity. Unfortunately, Terminal Server limits its activities to exporting imagery to a remote display and receiving remote input; it is not possible for a remote end-point to implement any of the required interactive behaviour. This means that NT cannot realistically be used by people who move between islands of connectivity. Since NT has a closed architecture, there is little scope, outside of Microsoft, for either improving the existing window system, or replacing it with a new one.

---

[8] Their task was significantly more difficult than just modifying the display routines, since NT did not support multiple users being logged in at the same time. Citrix had to add this functionality too.

[9] Despite the fact that NT 4's window system lives within the NT Executive, most of the terminal server code runs as a user-level process.

2.4 Other window systems

This section briefly describes some other, less common, window systems which have some notion of network distribution.

2.4.1 NeXTSTEP / OPENSTEP / Mac OS X

NeXTSTEP was an operating system with an integrated window system and GUI created for the NeXT line of workstations. The NeXT machines were based on a BSD-like operating system running above the Mach micro-kernel. *OPENSTEP* is a cross-platform version of NeXTSTEP, running on various Unixes and Windows NT. Apple have recently announced that NeXTSTEP is to form the basis of their new Macintosh server operating system, Mac OS X.

NeXTStep's window system is based upon Display PostScript[Hol90], meaning that it is completely device independent. In a similar manner to X, remote applications may communicate with the NeXT display host via a byte-stream protocol, and there are no interactive elements within the display sever. There is no support for mobile users.

2.4.2 $8\frac{1}{2}$

$8\frac{1}{2}$ is the window system of Bell Labs' *Plan-9* operating system[Pik91]. Plan-9 extends the Unix concept of special device files to all aspects of the operating system. Plan-9 filesystems may be local or remote, meaning that the whole operating system may be distributed. Furthermore, filesystem naming is local to a process—two processes may refer to different files (or devices) by the same name.

$8\frac{1}{2}$ provides applications with a number of special files, such as /dev/console and /dev/mouse which allow them to interact with the window system. Each process sees its own version of these files, hiding the multiplexing performed by $8\frac{1}{2}$. The /dev/console special file supports both simple text input/output, and more complex graphical output.

The $8\frac{1}{2}$ special files, like all Plan-9 files, may be accessed across a network, so remote users are supported. There is, once again, no means of re-directing a desktop session in progress, so there is no support for mobile users. Even if there was, all interactive behaviour must be supplied by the applications, so $8\frac{1}{2}$ is highly dependent on latency.

2.4.3 Berlin

The Berlin project is a work-in-progress window system intended to replace X on Unix machines[Ber98]. The system is based upon a monolithic window system server which exports a number of CORBA objects[OMG98a] to clients across a network. There is no explicit support for user mobility, but CORBA does support dynamic re-location of objects (see Chapter 5).

More importantly, the Berlin server *does* support interactive components, so it has the potential to be less latency-dependent than other systems.

2.4.4 $\mathcal{WS}$

In his Ph.D. dissertation, Barham[Bar96] describes a novel experimental window system, named $\mathcal{WS}$, which runs under the *Nemesis*[LMB+97, Ros95] multi-media operating system. $\mathcal{WS}$ is a *client rendering* system meaning that, rather than all rendering operations being performed by a single window system server, individual clients render their own imagery directly into the display's framebuffer. To prevent clients rendering to parts of the framebuffer that they do not own, a very-low level system implements access control: for each pixel on the display, a value containing a window number is stored; only clients with a matching window number are permitted to render to a given pixel. This can either be implemented in software, or hardware such as the Desk-Area-Network frame store[Bla95].

The design of $\mathcal{WS}$ is very closely tied to the Nemesis operating system, and much of its design would not be appropriate to other systems. It is also only designed to be used in a local-area network environment, and would perform as badly as other systems in the wide area.

2.5 Adding mobility support to existing systems

This section describes two different schemes which add support for users who are mobile within a single island of connectivity—a single local-area network—to existing window systems. There is no barrier to using either system in a wider area, but their performance becomes very poor if there is high latency or low network bandwidth.

The *Teleporting* system uses the standard X protocol, so it can only be used within an X environment; *VNC* uses a simple window system independent protocol, and implementations exist for a number of window systems. Both systems were developed at AT&T Laboratories Cambridge[10].

2.5.1 Teleporting

The idea behind the Teleporting[RBMH93] system is very simple, as shown in figure 2.3. The Teleporting proxy server acts as both an X server and an X client: to existing X clients, the proxy server appears just like a normal X server; to an X server, the proxy server appears just like a normal X client.

[10] Previously ORL, the Olivetti and Oracle Research Laboratory.

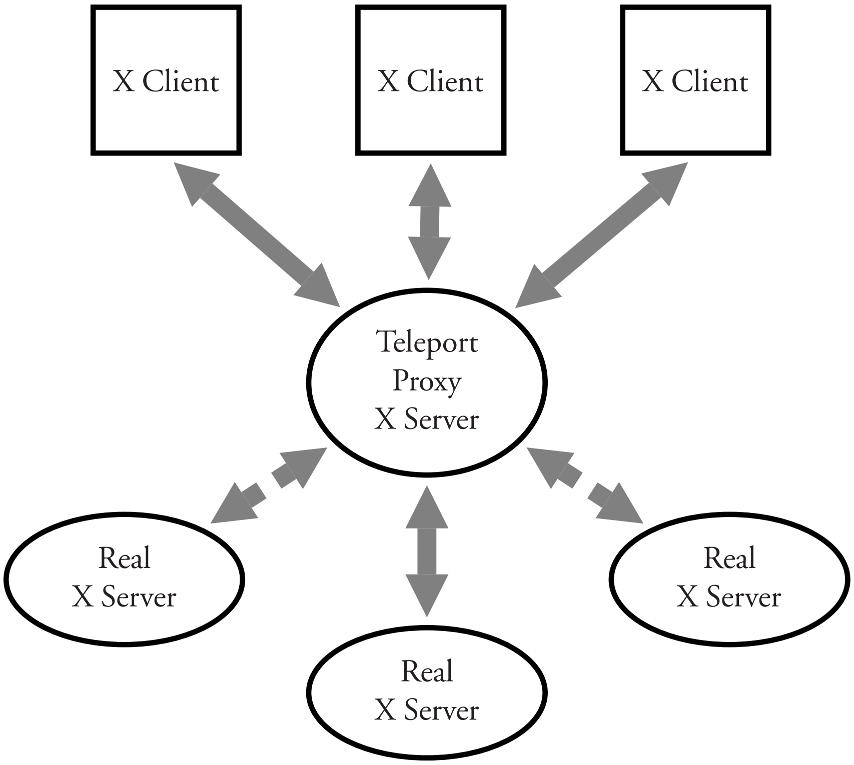


Figure 2.3: Teleporting system architecture

To create a Teleporting session, the user first starts an instance of the proxy server and runs their normal X clients, including window manager, within it. To teleport their session to a particular display, they instruct the proxy server (via typed commands or an Active Badge[WHFG92] system) to connect to the relevant X server. The proxy server opens a single large window containing all the sub-windows which are part of the session. The user may then interact with their session as if it was directly connected to the local display. They may even cut-and-paste data between a Teleport window and any other X window (including a different Teleporting session).

The user may teleport to a new location simply by instructing the proxy server to connect to a new display. The old Teleport window automatically disappears. The session need not be displayed at all times—it may be *dematerialised*. If the Teleport window is forcibly closed for any reason, the session is automatically dematerialised, so the session state is not lost.

Teleporting has to cope with displays with different colour capabilities and different sizes. Colour is handled rather naïvely—the proxy server presents itself as an 8-bits-per-pixel display with a fixed colour map, and attempts to render itself on whatever actual display it encounters. This causes problems if the display has more restricted colour capabilities, such as monochrome or 8-bit colour with a different fixed palette, leading to items which should be distinguished by colour being rendered in the same colour as each other. It is also unfortunate that if the display has better colour capabilities, like 24-bits-per-pixel true colour, clients are unable to take advantage of it.

X was not designed to expect that the display size could change at run-time, so clients may end up partially or totally off-screen when the user teleports. Teleporting adds new properties to the X root window which allow clients to find the current display size and name, but applications have to be modified to make use of them. The main application which understands the new properties is the *tpwm* virtual window manager (based on *tvtwm*), which allows the user to scroll around a large virtual desktop regardless of the size of their current viewing window.

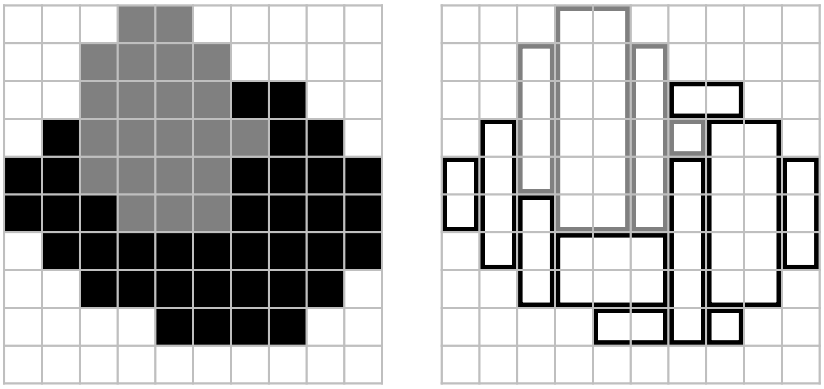
Teleporting has proved itself to be extremely useful in a local-area environment, especially when coupled with an Active Badge system. The use of the proxy server does, of course, double the network usage and the impact of latency, but in a normal LAN this effect is hardly noticeable. The desire to extend Teleporting to wide-area mobility, and to provide better support for colour and size differences was a major influence on the decision to build DAWS.

2.5.2 VNC

VNC[RSFWH98], the Virtual Network Computer system, grew out of the work on the Video Tile[ATT96] at ORL. The Video Tile is a very simple display device directly attached to an ATM (Asynchronous Transfer Mode) network. The Tile required a window system, but was not powerful enough to run an X server. Rather than creating a new, simpler, window system, the solution was to run a modified X server on a workstation connected to the ATM network, and send the pixels making up the display to the Tile. Only updates to the display were sent, but the pixel data was not compressed. A very simple protocol named RFB (Remote Frame Buffer) was designed to send the data.

The system was then extended to allow RFB to be displayed on a desktop machine, or within a Java-enabled web browser. The clients were no longer all on a very high speed network, so compression was added to the RFB protocol. There are several compression schemes, but they are all based on the idea of rise-and-run-length encoding (RRE), shown in figure 2.4. RRE is like run-length encoding in two dimensions—the pixels are represented as a series of rectangles of solid colour. RRE obtains reasonably good compression for most desktop displays with very little processing overhead, the notable exception being stipple patterns which encode very badly indeed.

VNC assumes that all displays support colour with either 8, 16, or 24 bits-per-pixel. Clients may request whatever colour depth is most appropriate for them, and the server sends pixel data in the correct format. Similarly, clients may request any of the types of compression, or none at all, and the server behaves appropriately. This means that it is very simple to write a VNC display client, since it only has to deal with those



The grid on the left shows the image to be compressed; the grid on the right shows the rectangles which result from the rise and run-length encoding. With coordinates (0,0) at top-left, the image is transmitted as: [white background (0,0)→(9,9)] [grey (3,0)→(4,5)] [grey (2,1)→(2,4)] [grey (5,1)→(5,6)] [black (6,2)→(8,2)], and so on…

Figure 2.4: Rise and Run-length Encoding used by the RFB protocol

parts of the protocol it wishes to. Clients are available for X, Windows, Mac, Java, and some other esoteric systems such as Linux SVGALib.

Since RFB is independent of any particular window system, it is possible for VNC clients to display the desktops from a number of different systems. The first server was for X, and was based on a modified X server. The X server does not attempt to render to a hardware display, but instead packages up all screen updates and sends them to clients. This leads to a very efficient VNC server.

Servers have also been written for Windows and the Mac. Unfortunately, unlike X, the source code to their window systems is not available, so the VNC servers have to attempt to intercept drawing operations, and read the rendered pixels from the screen. This approach is reasonably effective, but imposes far more of a processing burden on the server machine than the X version does. Since the pixels are first rendered to the server machine's display, then read by VNC, it is not possible to run multiple VNC sessions on a single machine.

It is also possible for stand-alone applications to directly output the RFB protocol, and a simple toolkit has been developed to aid this.

Like Teleporting, VNC allows display clients to disconnect without closing the session, so users are able to move from display to display, taking their sessions with them. Like all of the other systems supporting user mobility described in this chapter, VNC performs well in the local area, but falls prey to latency and lack of network capacity in a wider area. VNC also has no way to inform applications that they have changed display, so they are unable to adapt to their new situation.

2.6 Conclusions

None of the window systems presently available are flexible enough to support today's highly mobile users. Extensions such as Teleporting and VNC go some way to solving the problem, but they are limited by their rigid design to only work well within an island of connectivity.

All of the systems which permit remote access of any kind work by providing a *common interface* to all possible end-points. This approach certainly makes designing the system much simpler, but it means that only a narrow range of devices are supported well. The only system which goes any way towards adapting to different situations is NeWS, but the core NeWS server functionality is sufficiently complex that it is not suitable for simple devices. No existing window system is able to make best use of the full range of available devices, from simple network attached displays to high-powered workstations with specialist graphics hardware.

The question is therefore whether an existing system should be modified, or a completely new system should be developed. Modifying an existing system would have obvious advantages, since it would be far easier for users to start using the new system if it ran existing applications, and worked in a familiar manner. The only system which could realistically be modified is X, since it is the only popular system for which the source code is available. Experience of others, such as the MTab group at Xerox PARC mentioned on page 11, suggests that this would be extremely difficult, if not impossible. It is therefore worthwhile to investigate the design of a brand new window system that can address these issues properly.

> The arguments put forward in this chapter lead to the conclusion that to properly support users who are mobile in a wide area, it is necessary to create a brand new window system which is able to adapt to the user's environment as they move from place to place.

Unlike current window systems, the new system should not be rigidly split at a small number of points, but should allow a network connection to fall between any two functional elements. In this way, the system can be flexible enough to adapt to any environment. The remaining chapters of this dissertation describe and evaluate such a system.

3
DAWS Architecture and Framework

The previous chapter described the most common, and some of the less common, window systems available. None of those systems are suitable for the demands of highly mobile users. This chapter first outlines the major requirements of a new window system designed to address these demands, and then presents the design of an architecture which meets the requirements.

The architecture of DAWS consists of two major elements: a general-purpose low-level framework upon which distributed systems may be built, and the window system itself, built upon the framework. The full window system design can only be understood in the context of the framework, so this chapter mainly concerns itself with the details of the framework. The design of the window system design is presented in the next chapter.

3.1 Design requirements

This section explains the design requirements of DAWS[1], based upon the shortcomings of the systems described in chapter 2. Later sections describe how DAWS meets these requirements.

Mobile users

The driving force behind the creation of DAWS was the wish to support mobile users. The system should permit the user's desktop to follow them as they move, both within a single island of connectivity, and between islands, without compromising the system's interactive performance.

Adaptable and extensible at run-time

The requirement to adapt at run-time is largely a result of the requirement to support mobile users. The system should also be able to adapt to perform load-balancing, and

[1]The name 'DAWS' was picked to reflect these requirements.

to benefit from new hardware when it becomes available. There should not be a fixed set of DAWS services—it should be possible to extend DAWS' functionality at any time.

Heterogeneous environment

DAWS should run on a wide variety of hardware and operating systems. Furthermore, it should be possible for a running session to involve different types of machines. It should also be possible for elements running on one type of machine to migrate to another type of machine.

Specialist hardware

As well as running on a wide variety of general-purpose hardware, DAWS should be able to make use of specialist hardware, such as the clipping hardware mentioned in section 2.4.4, if it is available. DAWS should not, however, rely on any non-standard hardware.

Specialist input devices

DAWS should also support a wide range of different types of input device, such as video and voice recognisers, as well as the conventional mouse and keyboard.

Modular and distributed

DAWS should consist of fine-grained modules, rather than relying on a single monolithic server like X and NeWS. It should be possible for modules to be remote from one-another.

Personalisation

Like X and NeWS, DAWS should provide the mechanism by which GUIs are built, not the policy by which they operate. It should be possible to 'plug in' different visual appearances, and different management interfaces. Unlike X, aspects such as visual appearance should be inter-changeable without recompiling applications.

Precise image rendering

As with X, applications should be able to render exactly the pixels they wish, on a pixel-by-pixel basis. Most applications will not require this, however, so higher level interfaces should also be provided as standard.

Failure resilience

The system should be resilient to failures of individual elements within the system. In particular, it should not be susceptible to the sort of problems which X has with server grabbing, as mentioned on page 11.

No migration intelligence

Although it would be useful for DAWS to automatically adapt to form the best configuration at any time, given the user's current location and the computer system's current status, such intelligence is outside the scope of this dissertation. Other projects, such as AT&T Laboratories Cambridge's SPIRIT[ASH97, SWH98], address such concerns. DAWS should, of course, present a simple management interface to systems such as SPIRIT.

Scope for security

Security, in the form of both authentication and privacy, is required by a distributed window system. Security is a highly complex area, and it is not an aim of this project to produce a fully secure system. However, DAWS should be designed in a manner which permits such security features to be implemented.

3.2 DAWS Architecture Overview

DAWS meets the above requirements with an architecture based upon CORBA (Common Object Request Broker Architecture) [OMG98a] objects[2]. This section presents an introduction to the architecture, so as to provide context to later sections. The rationale behind the design is explained later in this chapter and in Chapter 4.

Figure 3.1 shows the top-level CORBA objects involved in a running DAWS system; for simplicity, it does not show either the large number of sub-objects used, or the means by which objects are created. The diagram shows the logical relationship between the objects; it is common for operations to short-cut the object connections shown in the diagram, and communicate directly with each other for added performance.

At the lowest level, the Display object manages either a physical or virtual display device. Physical displays are actual hardware displays; virtual displays may be windows within DAWS or another window system, or interfaces to other display protocols such as VNC. Different Display objects may support rendering functions of different complexities, via the *Resource* mechanism described in section 3.5.5.

Overlapping regions, such as windows and menus, are provided by Layer objects—one Layer per region[3]. Regions need not be rectangular; their shapes are represented in an efficient form detailed in section 4.5. Layers provide all of the drawing primitives required by higher-level entities, either directly or remotely, again via the Resource mechanism.

The LayerMgr object is responsible for managing the individual Layer objects, controlling their relative positions and their currently visible regions. Layers are reasonably

[2] The reasons for choosing CORBA are outlined in section 3.3.

[3] The term 'layer' is due to Pike[Pik83].

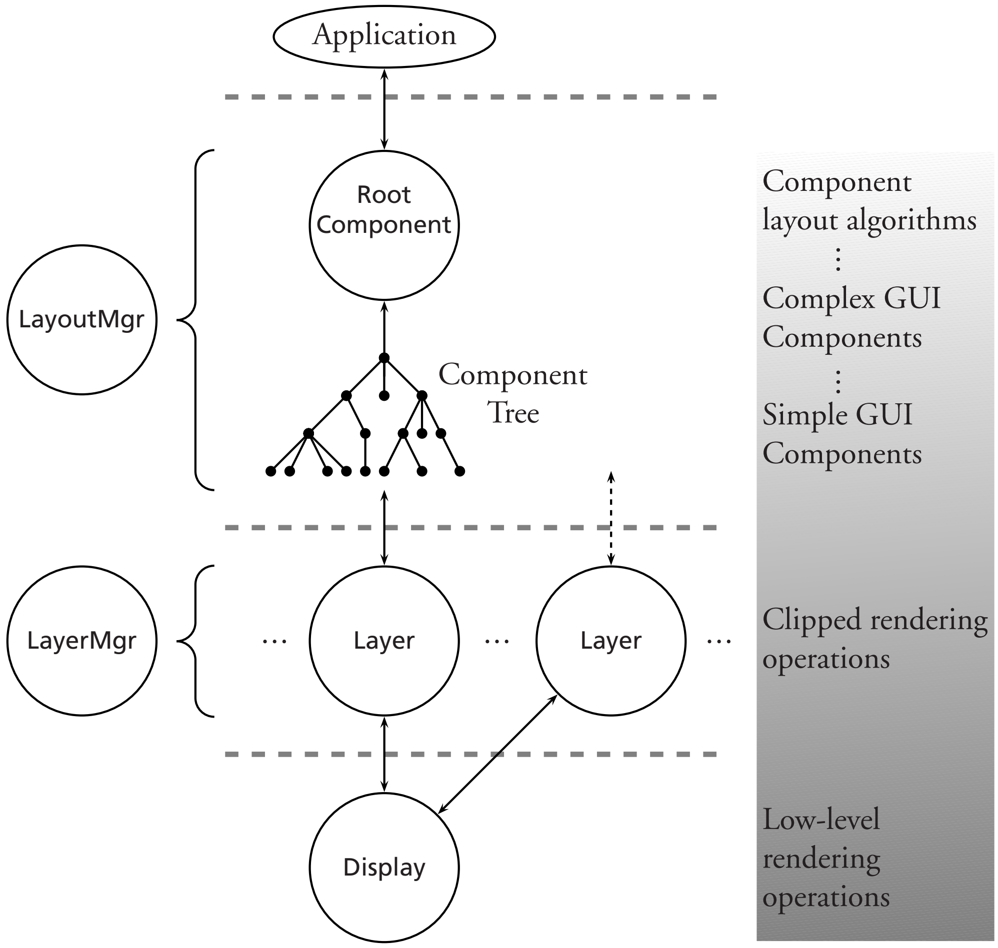


Figure 3.1: Overview of DAWS

low-level entities, and should therefore be close, in network terms, to the Display. The LayerMgr, on the other hand, only requires a low bandwidth connection to the Layers, and could even run in a different island to the Display and Layers.

Each Layer contains a tree of Component objects which are responsible for creating the appearance and interactive behaviour of the GUI. The specification and design of individual Components is, therefore, not really part of the window system at all, but part of the GUI built on top of it. The common Component base interface is part of the window system design, however.

The Component interface definition is very flexible—some Components arrange sub-Components in some manner; some are actual interactive GUI elements; others are a combination of both. The root Component is responsible for the layout of the whole Layer, and determines whether it is a window, a menu, or some other basic entity. The LayoutMgr object coordinates the different Components which perform layout.

Input management is performed by the same hierarchy of objects. Events originate at (or near) the Display object, and are sent to one or more consumer objects. At any given time there is a single active Layer, whose Components may receive input events. It is also possible for special objects (such as window managers) to intercept input events outside of the normal hierarchy. Since input events are often very susceptible to latency, only management of input is performed through the hierarchy—actual input events pass directly from the object creating the event to the event's consumers.

3.3 Object-Orientation and CORBA

This section justifies the use of an object-oriented framework in general, and CORBA in particular, as the framework for DAWS.

Firstly, the requirement for a modular and distributed system suggested that a distributed object framework should be used. Such a framework not only provides a standard means by which modules can be represented as objects, but also avoids the need to create a new communication protocol. This is important, because a large factor in X's inability to extend to support mobile users is that the X protocol is too rigidly defined; a protocol designed to allow greater flexibility would probably closely resemble a remote procedure call (RPC) or distributed object protocol. An alternative would be to create an ad-hoc system such as that employed by NeWS, where modules communicate with each other by module-specific protocols. It is likely that this would lead to an unmanageably complex system, given the far greater level of distribution involved.

A distributed object system, rather than an RPC system, was chosen because it is much simpler to manage state within an object-oriented system than within a purely procedure-based one. This is because an object handle encapsulates the context of an operation invocation as well as addressing information. In traditional RPC systems, the RPC handle contains just the addressing information, so all invocations must explicitly provide the context for the operation.

One problem with systems based upon remote procedure calls is that they are inherently synchronous. Performance is therefore tied much more to latency than to network capacity.

DAWS has been developed with the theory that a high performance window system *can* be built on top of a synchronous transport, provided multiple threads are used wisely, interfaces are designed carefully, and objects are placed so that high latency links only fall between objects when it is not problematic.

It would not, for example, be sensible to draw an image by repeatedly invoking an operation which plotted single pixels; an operation which plotted a large number of pixels in a single call would be provided instead.

3.3.1 Object framework requirements

The system requirements outlined in section 3.1 lead to a number of requirements on the object framework:

Interface based
Objects must be accessed through well-specified interfaces which do not depend on the objects' implementations. It is important to have well-specified interfaces, so that compile-time type safety is possible. Furthermore, the interface of an object must be independent of its implementation, so as to permit special-purpose implementations.

Heterogeneous
The framework must not rely on any particular hardware or operating system.

Location-independent
Clients of an object should not have to be aware of its location to use it. The system should, however, optimise communications so that calls to objects in the same address space or on the same machine are performed efficiently.

Object migration
It must be possible for objects to migrate from one location to another at run-time, so as to follow users. It must also be possible for an object to migrate to a different implementation of the same interface, to support special-purpose hardware. Migration of an object should be transparent to its clients.

Global scaling
The system must scale globally to cope with users moving in a wide area. In particular, it must not rely on storing information about object locations and similar meta-data in a central location.

3.3.2 Choice of framework

There are only three object-based middleware systems which are widely used at present: Microsoft DCOM (Distributed Component Object Model), Sun's Java RMI (Remote Method Invocation), and CORBA (Common Object Request Broker Architecture). All three are interface-based and support object location independence. The following evaluates how well each of them addresses the other requirements:

DCOM

At the time this work was started, DCOM[Mic96, Mic98a] was a purely proprietary system running only on Windows NT so it was immediately discounted. Since then, Microsoft have made some efforts to open the DCOM specification, and have it ported to other operating systems and architectures. Multi-platform support is still very limited, and DCOM has a number of other serious shortcomings[CHY$^+$97, MET98]:

- The object model is very awkward, probably due to DCOM's history as an extension of the non-distributed COM and OLE embedding mechanisms for office documents. The fundamental unit of distribution in DCOM is the *class*. Given a class identifier, the class can be used to create object instances. References to object instances themselves may only be stored using special *moniker* classes.

- DCOM provides no specific support for object migration.

- DCOM uses a reference-counting and pinging mechanism to control the deletion of objects. Whenever a new client attaches to an object, it increments the object's reference count; when it no longer requires the object, it decrements the reference count. To cope with abnormal client termination, clients ping the server at fixed two-minute intervals; if the server fails to receive three pings, the reference count is decremented. Although the system goes to some lengths to reduce the network impact of the ping mechanism, it is still a limit to how well the system can scale.

Java RMI

Java's Remote Method Invocation[Sun98], like the Java language itself, avoids the issue of platform heterogeneity by running within the Java Virtual Machine[LY96]; all platforms which have a JVM 1.1 (or higher) implementation support RMI.

Java's homogeneous bytecode allows Java objects to be sent by value through RMI, including the object's state and implementation. At first, this would seem to be a useful facility for implementing object migration. However, when an object is passed by value, it is *copied* rather than moved, so a migration mechanism would still have to be implemented. Unfortunately, the RMI protocol has no provision for object migration.

Other factors which led to the rejection of RMI include:

- Java is designed to be completely platform-neutral, and so has no means of directly accessing hardware devices. Functions written in other languages, such as C or C++ can be invoked through the Java Native Interface, JNI[Sun97a], but this would be inconvenient and inefficient for objects such as displays, where almost all of the functionality requires access to hardware.

- JVM implementations are not available for all platforms upon which elements of DAWS should be able to run, most notably specialist hardware such as the video tile[ATT96].
- At the time this project was started, Java was quite slow due to bytecode interpretation. This would have been a problem for low-level parts of the system, such as graphic rendering. Just-in-time compilers are now available for some platforms, so the problem is alleviated to some extent.
- RMI, like DCOM, makes use of a reference counting and pinging mechanism, limiting the system's scalability and flexibility.
- Most importantly, RMI ties the system to the use of Java (or at least the JVM). This is dangerous, as it may prevent the use of new future languages and technologies. Cross-language support of RMI is difficult if the target language does not have the same, or very similar, features as Java.

CORBA

CORBA, the Common Object Request Broker Architecture[OMG98a], is an open standard defined by the Object Management Group[OMG]. A large number of vendors provide implementations of parts of the standard, for many different systems.

The CORBA standard specifies an object model, an Interface Definition Language (IDL) and mappings from IDL to a number of languages, including C, C++, and Java. It also defines a class of inter-operable wire protocols, GIOP (General Inter-ORB Protocol), which allow different CORBA implementations to inter-operate. All fully compliant CORBA implementations must support IIOP, the Internet Inter-ORB Protocol, an instance of GIOP over TCP/IP. Clients written using one CORBA implementation can thus connect to servers written using another CORBA implementation.

Unlike RMI and DCOM, the requirement that objects may migrate *is* addressed by GIOP (and thus IIOP), so CORBA is a far more suitable choice of platform for DAWS. Also, CORBA has no built-in reference counting mechanism, leaving such concerns to the object implementation. Although this can mean extra complexity for the programmer, it allows object life cycles to be far more flexible.

CORBA was the obvious choice as the basis of DAWS.

3.4 A closer look at CORBA

This section summarises a number of aspects of the CORBA standard which influenced the design of DAWS. There are two main elements to the CORBA specification, the core standard[OMG98a] and the CORBA services[OMG98b], both of which influenced the design of DAWS. The core defines the facilities which all CORBA implementations should support; the services provide standard interfaces to a wide range of additional useful capabilities.

3.4.1 Object model

The first element of importance in the CORBA core is the object model[4]. It is a classical object model, in that clients send messages to objects. Each message identifies an object and a number of parameters; the object chooses a method to perform based on the parameters. Messages within CORBA are termed *requests*, and consist of a target object, an operation name, and zero or more actual parameters.

The *values* which may be stored in the parameters of a request must have a type specified in CORBA IDL—either a basic type or a constructed one. Basic types are values such as integers (of various sizes), floating point numbers (of various precisions), characters, strings, and object references; constructed types may be fixed-sized *arrays* or variable-length *sequences* of a single IDL type, structured records of items, or discriminated unions of different types.

Object references are values which denote a particular object. A particular object reference will always resolve to the same object, but an object may have multiple distinct object references which all resolve to it.

Request parameters may be declared as being input only, output only, or input-output. Operations may also return a single return result, which is essentially the same as an additional output parameter. Instead of returning successfully, having filled in any output parameters and the return value, an operation may raise an *exception*. In this case, the values of any output parameters and the return value are undefined. Exceptions are a specialised form of structured record, and may thus contain information about the cause of an error.

Object creation and destruction are purely the responsibility of the server—there are no standard client interfaces to such activities. Interfaces must have explicit support for operations which affect object life cycles.

3.4.2 The CORBA core

As well as defining the object model, the core specification is where IDL, GIOP, and the language mappings are defined. It also presents the idea of an *object adaptor* (OA), which is responsible for managing CORBA objects within a process. The CORBA 2.0 specification[OMG96] defined the Basic Object Adaptor (BOA). The BOA was not specified in sufficient detail to allow object implementations to be portable between different CORBA implementations. This was addressed by the CORBA 2.2

[4]This description is a summary of the first chapter of the CORBA specification[OMG98a].

specification[OMG98a] which introduced the Portable Object Adaptor (POA). There are, at present, few CORBA implementations which support the POA.

The architecture provides for a number of different object activation policies: objects which are always available, ones which are loaded on demand, and ones which are purely transient. Some ORBs make use of *implementation repositories* to manage the location and activation of objects within a domain. To use an object registered with an implementation repository, a client first contacts the implementation repository before being forwarded to the actual object. The implementation repository is necessarily a very ORB-specific entity, and so has no standard interface.

The CORBA core provides for both static and dynamic access to objects. With static invocation, both the client and server have compile-time knowledge of all IDL types and interfaces they are using. This allows compile-time type checking, and provides for efficient implementations. Use of static invocation involves passing the IDL definitions through an *IDL compiler*, resulting in a set of language *skeletons* and *stubs*. Implementations and clients then interact with this generated code.

The first level of dynamic behaviour is achieved with the `any` type. An `any` can contain a data item of any of the IDL base types, or a constructed type. Types are checked when attempts are made to extract the data from an `any` at run-time.

Clients which have no compile-time knowledge at all of the objects they are accessing are supported through the Dynamic Invocation Interface (DII). Similarly, servers can create objects at run-time with the Dynamic Skeleton Interface (DSI). Information about object interfaces may be dynamically found and set with the *Interface Repository*. DII and DSI have very limited use; they are mainly only used by object browsers, and bridges to and from other communication mechanisms such as DCOM.

3.4.3 CORBA services

There are three basic types of services specified in the CORBA services[OMG98b]. Some, like the *Security* service, are ORB-specific, in that they must be intimately linked with the ORB implementation. Others, such as the *Naming* and *Event* services, are ORB-independent: it should be possible for a client using one ORB to use a service implemented with another ORB. Finally, services like the *Life Cycle* service are design patterns whose implementations are application dependent.

The four services mentioned above are particularly relevant to the design and implementation of DAWS. The Naming service provides a database which maps from hierarchical names to object references, and thus provides a means for finding initial object references. Rather than registering all objects with the Naming service, which would place an unreasonable burden on a single entity, most objects are found (or created) via other objects. When a CORBA application is initialised, it is able to find a reference to a local root naming context, through which it can register objects or find objects it requires.

The Event service specifies a system through which generic events[5] (stored in anys) may be propagated between producers and consumers, mediated by an *event channel* object. The Event service provides one possible model by which input could be handled in DAWS but, as explained in section 4.3, a more rigidly-specified system is used for low-level events such as mouse movements, and a simpler event system is used for generic high-level events.

The Life Cycle service specifies generic interfaces which may be used to control object creation, deletion, and migration. The concepts behind it are ideal for the implementation of the adaptive behaviour required by DAWS. As detailed in section 3.5.4, DAWS in fact uses a modification of the standard Life Cycle interfaces.

The Security service is an extensive specification of how authentication and privacy should be provided within the CORBA framework, via cryptographic and other means. The Security service would provide the basis for a secure implementation of DAWS.

3.4.4 Choice of language and CORBA implementation

For most of the design of DAWS, the particular programming language and CORBA implementation used (and indeed the use of CORBA at all) is not important—the design decisions are independent of such issues. However, it is useful to discuss these choices at this point, since they provide context to some of the examples used in this chapter and Chapter 4.

The most significant programming language issue is that DAWS is not tied to the use of languages like Java that support portable object code. Since code portability is not guaranteed, there will be difficulties if the user moves to a location where implementations of objects they require are not available. This is not considered too much of a problem, since all low-level objects will generally be available everywhere that DAWS is available at all. Most higher-level objects will be less dependent on latency than low-level objects, meaning that they can run remotely from the user. Implementations of those few objects which really do need to be run within the same island of connectivity of the user could be installed manually, or written in a portable language such as Java.

All CORBA implementations should, in theory, inter-operate with each other, so it should have been possible to implement parts of DAWS with different ORBs and in different languages. However, there are still interoperability difficulties due to bugs and inadequacies of the specifications[Har98], and using a number of languages would have led to duplication of programming effort. Therefore, a decision was made to implement the whole of DAWS with a single language and ORB. Of course, this decision does not prevent future elements of the system being created with more diverse tools.

[5] The Event service also permits strongly-typed events, but that makes the whole system much more awkward. There are very few Event service implementations which support typed events.

The two main programming languages currently used for CORBA development are Java and C++. Since one of the primary aims of DAWS was to evaluate whether a CORBA-based window system could perform well enough to satisfy users, it made sense to choose C++ as the implementation language. Java is (or at least was before just-in-time compilers arrived) markedly slower than C++.

omniORB2[Lo98, LP98], from AT&T Laboratories Cambridge, was selected as the CORBA implementation for two main reasons. Firstly, it is one of the fastest ORBs available[BT98]. Secondly, none of the ORBs available at the start of the project provided useful support for the `move()` operation of the Life Cycle service. omniORB's source is freely available[6], and it was developed locally, so it was the best available platform upon which object migration could be built.

omniORB runs on many Unix variants, Windows NT, and a number of other operating systems. It does not yet support the POA, and so uses the older BOA specification. It also lacks an implementation repository, so only simple object activation modes are possible. All CORBA objects in omniORB are notionally *transient*[Hen98], but it is possible to re-create an object with the same IOR as a previous object, so some of the properties of *persistent* objects are available.

3.5 DAWS Framework

This section details the framework upon which DAWS is built. Although the framework was designed to form the basis of a window system, it is flexible and powerful enough to be useful for many other applications. The framework consists of:

- A set of policies concerning semantics of interfaces and the use of generic interfaces.
- The interface definitions and implementations of various infrastructure objects.
- A number of standard interfaces from which many objects derive.
- Library functions which simplify the implementations of DAWS objects.

This section explains the reasoning which led to this design.

3.5.1 Syntax and semantics

CORBA IDL allows the *syntax* of types and operations to be specified. It does not, understandably, allow *semantics* to be specified, other than informally within comments. This could, potentially, lead to problems, especially since a primary feature of DAWS is that objects are allowed to migrate to different implementations. Imagine, for example, the following simplified interface for receiving mouse events:

[6] Under the GNU Public Licence and GNU Library Public Licence.

```
interface MouseClient {
  void MouseData(in short x, in short y, in unsigned short buttons);
};
```

What do the arguments to MouseData() mean? How is the button state stored in the unsigned short? Are the coordinates absolute or relative to the last reported position? Both types of coordinate reporting may be useful: most applications would find it most useful to receive absolute positioning; an application which used the mouse to rotate around a three-dimensional view would prefer relative coordinates.

DAWS imposes the policy that the semantics for a given interface are fixed, defined by comments within the IDL file. The semantics of inherited interfaces are also maintained.

So, for example, if the above MouseClient interface reports absolute coordinates, a RelativeMouseClient interface must *not* be defined as

```
interface RelativeMouseClient : MouseClient {
  // MUST NOT inherit the MouseData() operation but
  // change its semantics
};
```

but should declare a completely unrelated interface (possibly with the same operation name).

A related issue is the naming of interfaces so as to avoid clashes. Just as it is important that the semantics of an interface are not intentionally changed, it is essential that a particular name is not *accidentally* used with different meanings in different places. It is relatively easy to avoid this sort of difficulty by qualifying all names with some globally agreed prefix, such as DNS names, in a similar manner to Java's package naming scheme[GJS96].

3.5.2 Creating objects

When figure 3.1, showing the architecture of DAWS, was introduced, the means by which objects are created was not explained. The most common method of creating a CORBA object is to use a *factory* object. A factory object simply has an operation which creates new objects in the factory's address space, and returns their object references. The two questions when using factories are whether the factories are generic or specialised, and how the factories themselves are found.

Generic

```
interface GenericFactory {
  ...

  Object create_object(in Key k,
                       in Criteria crit)
    raises (NoFactory,
            InvalidCriteria,
            CannotMeetCriteria);

};
```

Specific

```
interface WindowFactory {
  exception CannotCreate { string reason; };

  Window CreateWindow(in string title,
                      in short width,
                      in short height)
    raises (CannotCreate);

};
```

Figure 3.2: Interface definitions for generic and specific factories

Generic and specific interfaces

A generic interface is one which may be used for a wide range of purposes, via the use of the any type; a specific interface is tailored for a particular purpose, with hard-coded types.

Many of the definitions in the CORBA services specification, including the Life Cycle and Event services, make use of generic interfaces. This does allow a single interface to be used over a wide range of subject types, but it also negates one of the primary advantages of using an interface definition language as opposed to a bytestream-based system—that of compile-time type safety. It also tends to make both implementation and client code more complex.

Consider creating a hypothetical Window object with a particular title and size, using both approaches. Figure 3.2 shows the GenericFactory interface from the Life Cycle service, and a specific-purpose WindowFactory. (The complete IDL for the Life Cycle service is shown in Appendix A.) Figure 3.3 shows the C++ code that a client would use to create a Window object with each. The example using the GenericFactory is based upon the recommendations in the Life Cycle service specification; other implementations using the same interface would be possible.

Not only is the implementation using the WindowFactory far shorter and simpler to understand, but it also benefits from compile-time type checking. If, for example, the window title was supposed to be a wstring rather than a string, the GenericFactory client would only receive an error at run-time (or may even crash the GenericFactory implementation) while the WindowFactory version would receive a compiler error.

Throughout the framework, and the DAWS system itself, generic interfaces have been avoided wherever possible.

Generic

```
Window_var win; // The Window to be created

CosLifeCycle::GenericFactory_var gf;
// Get a GenericFactory...

CosLifeCycle::Key       key;
CosLifeCycle::Criteria  crit;
CosLifeCycle::Criteria  init;

key.length(1);
key[0].id   = (const char *)"Window";
key[0].kind = (const char *)
              "object interface";

init.length(3);
init[0].name    = (const char *)"title";
init[0].value <<= CORBA::Any::from_string
                  ("Example Title", 0);
init[1].name    = (const char *)"width";
init[1].value <<= (CORBA::Short)200;
init[2].name    = (const char *)"height";
init[2].value <<= (CORBA::Short)100;

crit.length(1);
crit[0].name    = (const char *)
                  "initialization";
crit[0].value <<= init;

try {
  CORBA::Object_var obj;
  obj = gf->create_object(key, crit);
  win = Window::_narrow(obj);
  if (CORBA::is_nil(win)) {
    // The object had the wrong type...
  }
}
catch (CosLifeCycle::NoFactory &ex) {
  // ...
}
catch (CosLifeCycle::InvalidCriteria &ex) {
  // ...
}
catch (CosLifeCycle::CannotMeetCriteria &ex) {
  // ...
}
```

Specific

```
Window_var win;

WindowFactory_var wf;
// Get a WindowFactory...

try {

  win = wf->CreateWindow("Example Title",
                         200, 100);

}
catch (WindowFactory::CannotCreate &ex) {
  // ...
}
```

Figure 3.3: Code to create a Window object with generic and specific factories

Finding factories

The ability to create new objects using factory objects is only useful if it is possible to find the object reference of a suitable factory. This is especially important when specific-purpose factories are being used, as there are far more factory objects present in the system, and it is not sufficient to find the single factory which is looking after a particular address space.

There are two essential requirements on a factory location system:

- Finding a factory which can create an object of the required type.
- Finding a factory in a particular physical location.

One approach would be to register all factories in the Naming service. This would avoid the necessity of creating a special-purpose factory database, but would have a number of disadvantages. Firstly, the requirement to be able to locate factories by a combination of two criteria would lead to an awkward naming scheme. It would also be difficult to perform look-ups on partial location information, probably requiring multiple requests to the Naming service. Another concern is that the choice of object used to perform factory look-up forms a crude choice of location—at least to island granularity—so the location domains would be determined by the placement of the Naming service objects. This would possibly limit the flexibility of the location strategy.

Instead of using the Naming service, DAWS borrows the idea of a *FactoryFinder* object from the CORBA Life Cycle service, although the standard FactoryFinder interface is not used. The standard FactoryFinder does not rigidly specify how factories to be found are named; the DAWS FactoryFinder is more specific, using a naming scheme based upon Internet DNS names and address space identifiers. Full details of DAWS' FactoryFinder interface and its implementation are given in section 3.5.4.

3.5.3 Controlling object migration and deletion

Once an object has been created by a factory, there are other aspects of its life cycle which must be controlled: the DAWS design requires that objects migrate from location to location, and it must be possible to delete objects which are no longer required. All DAWS objects which are to be controlled in this way must derive from the LCObject interface (shown later in figure 3.4) which defines Move() and Remove() operations. Both operations require some knowledge of the activities of the object, so they must be provided by the object implementation, rather than by the system.

Implementing the Remove() operation is relatively simple: the object un-registers itself from any services it has registered itself with, tells the ORB it is no longer active, and then deallocates all storage it is using[7].

The Move() operation consists of two phases, both of which are troublesome. In the first phase, the complete state of the running object must be transferred to the new location; in the second phase, all clients of the object must start using the new object. The second phase is considered in depth in Chapter 5.

[7]There are, in fact, a few other subtleties to object deletion due to concurrency control issues, explained in Chapter 5.

There are two basic methods by which the first phase, state transfer, could be performed. One option, advocated by some[Dea98], is to somehow take a snapshot of the running object, and transfer that state. Various process migration systems permit this[JLHB88, SJ95, SH97], as, to some extent, does Java's object serialisation[Sun97b, BHR$^+$98, Obj97]. This sort of approach would be problematic for DAWS, not least because it would be difficult to reconcile with the requirement that objects may move to different implementations.

It is also usually the case that a running program has a large quantity of run-time state which is not really part of the logical object's state, but an artifact of the implementation.

Even if there was no problem with different implementations, a large volume of irrelevant state, would be transferred under such a scheme. Examples include temporary local variables within functions, pointers within linked lists, and exact details of running threads. Finally, there are serious practical difficulties to implementing a general state migration scheme without extensive operating system and language support.

The second option is for objects to transfer their own state when they are instructed to migrate[HL82][8].

DAWS takes the view that an object should be responsible for transferring its own state upon a call to its `Move()` operation. This way, only the relevant state need be transmitted, and it may be formatted in an implementation-independent manner.

The art of designing a good state transmission format which reveals no implementation details is similar to designing a file format, although often simpler as most objects have reasonably simple state. The best way to transfer state is an issue which must be evaluated for each type of object, and particular objects may even support several different methods. For example, an object editing a file could migrate its state to a machine which had access to the same filesystem simply by passing the name of the file and the current editing position; to migrate to a machine with a different filesystem, it would also have to transfer the contents of the file by some means.

State transfer when moving a DAWS object can occur in two ways. The first, and most common, is that the factory object has a *clone* operation which takes the object's state in its arguments. The factory is then able to create an implementation object with the correct state. The second is to use an operation (or several operations) of a newly-created object to set its state, according to the state of the existing object. As explained

[8] In fact, it is also possible to use a hybrid option in which the system automatically transfers state during migration, but the programmer explicitly tags which state should be transferred[Pop96].

in Chapter 5, the object migration system used by DAWS requires some special state to be transferred at object creation time, so a clone operation is always required, even if it is not used to set all of an object's state.

3.5.4 Details of the DAWS Life Cycle services

The DAWS framework services mentioned so far correspond directly to the services provided by the CORBA Life Cycle service, although there are a number of differences in the details of the interfaces. The DAWS Life Cycle interfaces are shown in figure 3.4; The IDL for the standard CORBA Life Cycle service is shown in Appendix A for comparison. If it was deemed necessary, it would be possible to wrap the DAWS interfaces in the standard Life Cycle interfaces.

The DAWSLifeCycle module has a number of simplifications compared to the standard CosLifeCycle module, which make interfaces of other objects and object implementations simpler:

- As explained in section 3.5.2, there is no GenericFactory—each DAWS object has its own special-purpose factory.
- Simple strings, rather than CosNaming::Names, are used to name entities.
- The FactoryFinder has an integral management interface.
- There are no Criteria sequences.
- The LCObject interface, the equivalent of the CosLifeCycle::LifeCycleObject interface, has no copy operation.

The following sections explain the interfaces in detail. It is not necessary to understand all of these details to follow the flow of the argument, so it is safe to skip to section 3.5.5 at this point.

The FactoryFinder

As stated earlier, the FactoryFinder is a database used to map object kinds to factories for those objects. There should be at least one FactoryFinder for each island within the DAWS system, meaning that specifying a particular FactoryFinder implicitly specifies a crude form of location. FactoryFinder objects themselves may be found through a number of sources, most commonly either the Naming service or a stringified object reference written to a file.

Factories are found using a combination of a sequence of Location strings and a Kind string. The Kind string is the Interface Repository identifier (in OMG IDL format) of the object which will be created by the desired factory. To find a factory for Window objects, the Kind string would be something like '`IDL:Window/1.0`'.

```
module DAWSLifeCycle {
  typedef string Kind;       // A kind is an interface repository id
  typedef string Location;  // Format is some.host.name:address
  typedef sequence <Location> Locations;

  exception NoFactory     { };
  exception NoLocation    { };
  exception NotMovable    { };
  exception NotRemovable  { };
  exception AlreadyAdded  { };
  exception CannotCreate  { };

  typedef Object Factory;
  struct FactorySpec {
    Location l;
    Factory  f;
  };
  typedef sequence <FactorySpec> FactorySpecs;

  interface FactoryFinder {
    FactorySpecs FindFactories(in Locations ls, in Kind k)
      raises (NoFactory);

    Time::Seconds AddLocation(in Location l)
      raises (AlreadyAdded);

    void LiveLocation(in Location l);

    void AddFactory(in Location l, in Kind k, in Factory f)
      raises (NoLocation, AlreadyAdded);

    void RemFactory(in Location l, in Kind k) raises (NoFactory);
    void RemLocation(in Location l)           raises (NoLocation);
  };

  interface LCObject {
    void Move(in FactoryFinder f, in Locations l)
      raises (NoFactory, NotMovable);

    void Remove() raises (NotRemovable);

    readonly attribute Location CurrentLocation;
  };
};
```

Figure 3.4: Interface definitions for the DAWS Life Cycle service

Each element of the `Locations` sequence may have one of three forms. The most specific form contains a host name and a string identifying an address space within the host, separated by a colon. The address space string may have any form, but on Unix-based platforms the process identifier is used[9]. An example is '`marrow:1234`'. Location strings of this form only match the exact location specified. The next most specific form contains just the host name, without the address space identifier. This form matches all address spaces on the specified host. The final form of location is the special string '`ALL`' which matches all locations registered in the FactoryFinder[10].

The `FindFactories()` operation returns a sequence containing a `FactorySpec` structure for each factory of the requested kind found at the specified locations. The sequence has the same order as that in which the locations were specified, so factories are found in order of preference. This means that sequences such as {'`marrow:1234`', '`marrow`', '`ALL`'} may be used to find factories in a particular address space or, failing that, any address space on the same machine or, failing *that*, any of the machines known to the `FactoryFinder`.

When a new process providing DAWS factories starts, it must acquire a reference to a `FactoryFinder` object from somewhere. It then registers its location string using the `AddLocation()` operation. Once the location has been added, each factory supported by the process is specified with the `AddFactory()` operation. `RemFactory()` removes a single factory from the database; `RemLocation()` removes all factories at the given location, and the record of the location from the database.

The `FactoryFinder` has no way to force processes to call `RemLocation()` before they exit, so it implements a low-rate pinging mechanism. `AddLocation()` returns a time in seconds; the factory process must call the `LiveLocation()` operation at least this often, otherwise the location will be deleted. Unexpected loss of a factory is a very soft failure since the `FindFactories()` call always returns a sequence of possible factories. The client iterates through this sequence until it finds a working factory, so it does not matter if some of the factories have ceased to exist[11]. This means that the ping time can be quite long.

FactoryFinder implementation

It would have been possible to use an existing general-purpose database to implement the FactoryFinder. However, the requirements on the database are very simple, and the

[9]On Linux, each thread within a process receives its own process id. Care must be taken to always use the id. of the parent thread when referring to a particular process.

[10]The empty string may seem a more obvious choice for matching all locations. However, using a non-empty string means that client bugs which cause them to accidentally use an empty string are more likely to be caught. There may be a problem if there is a machine in the local domain with the name 'ALL', but since the location string is case-sensitive and DNS names are not, a string of '`all`' could be used to refer to that machine.

[11]The example code in figure 3.8 shows this iteration.

number of items stored in it, even in a full production system, would be of the order of thousands at the most. A special-purpose implementation was simpler to design, and is probably more efficient.

The design is modelled upon the work of Birrell et al[BJW87][12]. The entire working database in stored in memory in an open hash table. A pointer to each factory entry in the database is stored at three hash table positions—one hashed by kind and full location, one hashed by kind and host, and one hashed just by kind. This means that all types of look-up may be performed very quickly. A second, much smaller, open hash table is used to store details about locations. Thread control is performed using a multiple-reader, single-writer interlock[Bac97], so many clients may read from the database concurrently, but database modifications receive exclusive access.

The database must be persistent, so all operations are logged to a *redo* log file. Every so often (currently every thirty minutes), the log file is re-processed so that only currently active locations and factories are stored. When the FactoryFinder is re-started, it re-creates the hash table from the log, and also ensures that it uses the same FactoryFinder object reference as before.

LCObject

The LCObject interface is the equivalent of the standard Life Cycle service's LifeCycleObject interface. All objects which may migrate derive from LCObject. There is no requirement within DAWS for objects to be able to copy themselves, so the LCObject interface does not have a copy operation. To use the DAWS framework within a system which *does* require such behaviour, a new interface derived from LCObject would be provided.

The Move() operation takes the object reference of a FactoryFinder, and a sequence of location strings. The object implementation is responsible for finding a suitable factory with the FactoryFinder, based on the given locations. This scheme is used, rather than having the caller pass in a sequence of factories, since the object's clients may not know its fully derived type. For example, a client may hold an object reference of type Layer, although the object in question is actually of type LayoutLayer, which is derived from Layer. If the client was to find a sequence of LayerFactories, the object would be unable to use them as migration targets, since it would actually require LayoutLayerFactories.

The Remove() operation is self-explanatory. The CurrentLocation attribute is useful for finding factories at the same location as an existing object.

[12]The main difference in this design is that a single file is used for both the checkpoint file and the transaction log.

3.5.5 The Resource mechanism

The abstraction of objects being created by factories is very powerful, and can be applied to most entities within a window system—windows and buttons are obvious examples. However, there are some objects for which the semantics of factories do not make sense: factories imply that a large number of objects of a particular type may be created with a single factory. This is not true in some cases, especially with objects representing hardware devices such as mice and keyboards. Another important issue is that, since DAWS is required to adapt to special-purpose hardware, a flexible means of extending interfaces is required.

Take, for example, the `Display` object which forms the lowest level element of the DAWS architecture (see figure 3.1). The `Display` object may either be representing an actual hardware display of some kind, or may be a virtual display within a window. This means that in some circumstances, many `Display` objects could be created at a particular location, but in others, only a single one would be permitted. It is reasonable to use the normal factory mechanism in this case, simply throwing the `CannotCreate` exception if an attempt is made to create a second `Display` object in the case of a hardware display.

It is not so simple when the facilities offered by the `Display` object are considered. With hardware displays, the hardware may be a reasonably powerful general-purpose workstation, a very thin device such as a video tile[ATT96] with very little local processing power, or a special-purpose display with hardware support for some rendering operations and high-level operations such as clipping[Bla95]. Furthermore, most, but not all, displays will support some form of pointing device; fewer will support some form of keyboard input; some may have more exotic input devices. For displays lacking input devices, other devices on the network will provide them. Virtual displays will come in various incarnations, with different ranges of functions provided by the underlying window system.

Higher-level elements of DAWS should be insulated from having to know about such issues. The system must also continue to function correctly when objects migrate between devices with different capabilities. The following sections discuss different ways in which this could be achieved.

Factory abuse

It would be possible to use the factory mechanism to create objects for all of the situations described above, just as it is with physical displays—using a factory which responds to its first call by creating an object of the relevant type, and then throws `CannotCreate` exceptions to all further object creation requests, until the original object is removed. As well as being a somewhat ugly abuse of the factory system, this approach would complicate another desired feature: the objects in question would be able to support multiple clients, so they would have to be registered somewhere so other clients could find them.

Another similar possibility would be to have factories which, rather than creating new objects, returned the references of existing objects. This would avoid the previous scheme's need to register objects, but would mean that some factory objects had different semantics to others. Not only would this be confusing, but it also breaks the DAWS rule that semantics for an interface are fixed.

Inheritance

Another approach would be to use inheritance to aggregate the interfaces of all devices and facilities that were part of the physical display at a particular location. A particular display object's interface could, for example, multiply-inherit from the mouse handler, keyboard handler, and drawing interfaces, as well as the base display interface. Then, clients of the display would be able to access the operations of all of the interfaces they required. By performing such 'mixin' inheritance near the leaves of the inheritance tree[HR93], different display objects can support different sets of facilities.

At first sight, this seems like a sensible idea. However, trouble arises if the `Display` object is migrated to a different type of display with lesser capabilities, such as one without a keyboard. The new display object *has* to support the keyboard interface, because it must have exactly the same interface as the original object. This is the case even if there were no clients to the original display actually using the keyboard interface.

The new display could act as a proxy to a real keyboard object in a different location, but this would still require extra processing from the display, and would not permit new devices to be added to some displays without adding them to others.

Another difficulty with aggregating interfaces via multiple inheritance is that IDL does not permit naming clashes between inherited interfaces. If, for example, both the mouse and keyboard server interfaces had an operation named `AddClient()`, it would not be legal to create an interface derived from both interfaces.

Objects with multiple interfaces

Yet another approach, which is used by a number of systems[RLML86, HR93, Cra86], is for each object in the system to support a number of distinct interfaces. Clients are then able to use a *rotate* operation to switch between the interfaces of an object to which they hold a reference. This approach avoids clashes between the name spaces of different interfaces, but, like the inheritance scheme, it cannot cope with objects migrating between sites with different facilities.

In addition to this difficulty, there is the fundamental problem that CORBA does not support objects which have more than one IDL-defined interface, unless DSI is used. Forcing widespread use of DSI would circumvent the majority of compile-time type safety.

The solution

These complications are all avoided with the novel Resource and ResourceFinder interfaces, shown in figure 3.5, which permit objects to have an arbitrary number of sub-objects associated with them. This is superficially similar to providing multiple interfaces to a single object, but it no longer prevents objects from migrating between sites with different built-in capabilities. It is also similar to a proposed extension to Java's Beans system[Sun97c] which would permit a single 'bean' to have multiple associated Java objects. That proposal is intended to facilitate code re-use, rather than to enable object extensibility.

> The fact that each Resource is a first-class CORBA object means that Resources can migrate independently of the top-level object of which they are conceptually a part.

Resource sub-objects have interfaces derived from the Resource interface. Resources are found through ResourceFinders; every time FindResource() is called with the same kind argument on a particular ResourceFinder, it returns the same object reference. ResourceFinders are not usually stand-alone objects—other interfaces, such as the Display interface, are derived from ResourceFinder. Through this mechanism, different displays may support different sets of sub-objects.

If an interface derived from ResourceFinder migrates, all of its associated Resources must move with it. As with the base DAWS objects, Resources must be responsible for moving themselves since only they may know what state, if any, they have. A design pattern for the steps involved in migrating an object derived from both LCObject and ResourceFinder are shown in figure 3.6. The grey areas indicate when the various objects are in existence; the darker areas show when a thread is executing within an object. The steps are explained in more detail below:

1. Move() is called on the original object.

2. The object finds a suitable factory with the FactoryFinder given as an argument to Move(), and calls the factory's clone operation. One of the arguments to the clone operation is a sequence of ResourceSpec structures, listing all of the Resources which have been issued by the original object.

3. The factory iterates through the ResourceSpecs, checking that its object can support all of the listed Resources. If it cannot, a NoResource exception is thrown to the original object.

4. The factory creates a new object with a copy of the original object's state.

```
module DAWSLifeCycle {
  exception NoResource    { };
  exception ResourceBusy { };
  interface ResourceFinder;

  interface Resource {
    void RFMoved(in ResourceFinder rf) raises (NotMovable);
    void RFRemoved()                   raises (NotRemovable);
    readonly attribute ResourceFinder  rfinder;
  };

  interface ResourceFinder {
    Resource FindResource(in Kind k)
      raises (NoResource);

    Resource CloneResource(in Kind k, in omniLifeCycleInfo li)
      raises (NoResource, ResourceBusy);
  };

  interface ResourceFactory {
    Resource CreateResource(in Kind k, in ResourceFinder rf)
      raises (NoResource);

    Resource CloneResource(in Kind k, in ResourceFinder rf,
                           in omniLifeCycleInfo li)
      raises (NoResource);
  };

  struct ResourceSpec {
    Kind     k;
    Resource r;
  };
  typedef sequence <ResourceSpec> ResourceSpecs;
};
```

Figure 3.5: Interface definitions for the DAWS Resource mechanism

5. Before returning the new object reference, the factory iterates through the ResourceSpecs again, calling RFMoved() on each resource, giving the newly created object as the ResourceFinder argument.

6. Upon a call to RFMoved(), a Resource implementation calls CloneResource() on the given ResourceFinder object. This operation creates a new Resource object ready to migrate to. (The omniLifeCycleInfo argument is explained in Chapter 5.)

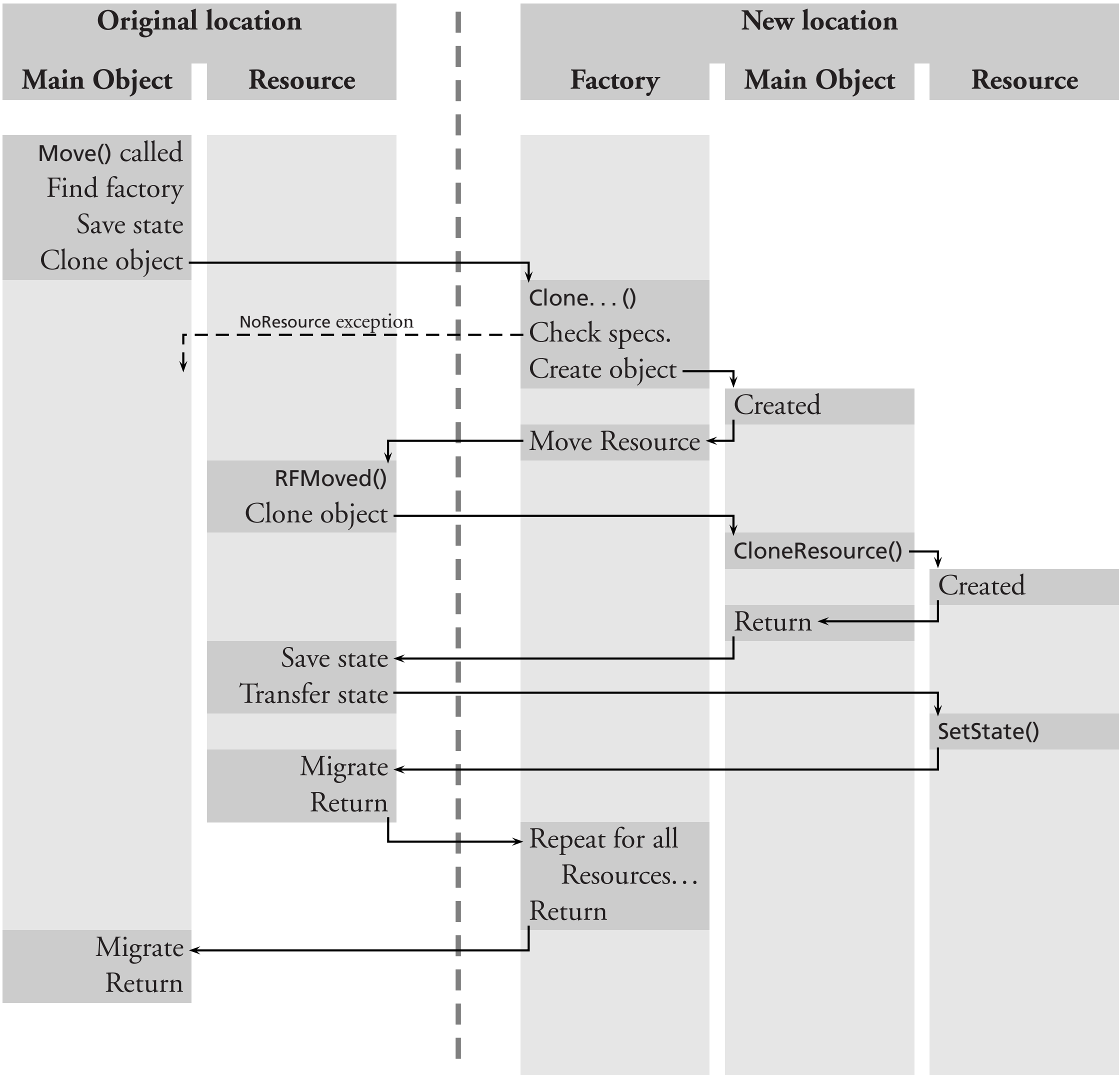


Figure 3.6: Steps involved in moving an object and all of its associated Resources

7. If the Resource has any state, it is copied using one or more operations of the newly acquired Resource object.

8. The original Resource object migrates itself to the new Resource object with a call to the ORB.

9. Once all Resources have been moved, the factory returns the new LCObject to the original object.

10. The original object migrates itself to the new object with a call to the ORB.

```
interface Display : DAWSLifeCycle::LCObject,
                    DAWSLifeCycle::ResourceFinder {
  ... Some operations ...
};

interface SimpleDraw : DAWSLifeCycle::Resource {
  void PlotPixel(in Coord x, in Coord y, in Colour c);
};

interface LineDraw : DAWSLifeCycle::Resource {
  void DrawLine(in Coord x1, in Coord y1,
                in Coord x2, in Coord y2,
                in Colour c);
};
```

Figure 3.7: Example interfaces for a Display object and two drawing Resources

Since these steps require a number of nested calls between remote objects, problems will arise if one of the machines involved fails part-way through the process. If the object's original location fails, the object will be lost, and any partial state created at the new location will be cleared away when the factory attempts to return to the original object. If the destination location fails, then some of the Resources may have been migrated, while others and the top-level object have not. Therefore, when Resources are migrated, none of their state should be discarded from the original location, until the entire object has successfully migrated. This means that, if the new location fails, the original Resources can be restored by their top-level object.

Resource factories

The ResourceFinder mechanism solves the problem of moving an object to a location with lesser capabilities, as long as only Resources supported by the new location were in use at the original location. The case where there were clients using Resources *not* supported by the new location is partially solved through the use of the ResourceFactory mechanism.

ResourceFactory objects are used for Resources which do not have to run at the same location as the ResourceFinder they are associated with. As an example, figure 3.7 shows the interfaces of a simple display object, and two drawing interfaces[13].

The SimpleDraw Resource must be implemented within the same address space as the Display, since it must have direct access to the display's frame buffer. On the other hand, there are two possible implementations of the LineDraw Resource:

[13]These interfaces are not actually used in DAWS; they are used here for simplicity.

1. A direct implementation which renders straight into the frame buffer, as the SimpleDraw Resource does.
2. An indirect implementation which renders via the SimpleDraw Resource.

The first approach would clearly be the most efficient, but would add to the complexity of the Display implementation. The second approach would be less efficient, but would allow the Display to run on less-capable 'thin' devices. Drawing simple lines is not a particularly complex operation; applying the same ideas to advanced rendering operations, such as curve drawing and three-dimensional projections, would be more beneficial.

ResourceFactory objects are registered with a FactoryFinder just like other factories. When a ResourceFinder is asked to find a Resource which it is unable to support itself, it tries to find a ResourceFactory for that kind of Resource elsewhere. If it does find a factory, it creates a Resource there, and keeps a reference to it. Any future requests for the same kind of Resource are met with the same object reference. A framework library provides this support, so implementation code does not need to deal directly with the FactoryFinder and ResourceFactory objects, or with storing the object references of all non-local Resources.

When a ResourceFinder which has made use of remote Resources migrates to a new location, it attempts to move those Resources to the new location, or at least to a location close to it, just as with local Resources. It is also necessary to augment step 3 in the migration steps described above to check for the availability of ResourceFactories if local Resources are not available.

Migration difficulties

If an object's Move() operation is called, and none of the factories it finds can create objects supporting all of the required Resources, either natively or via ResourceFactories, then there are two possible actions it can take. Firstly, it can throw a NoFactory exception, and remain in its old location. Secondly, it can remove the offending local Resources and migrate anyway. The next time a client attempts to access such a resource, it will receive an OBJECT_NOT_EXIST exception. Whether or not the client can cope gracefully with this situation depends whether the Resource was essential to the client's operation, or just useful.

Use of the Resource mechanism

The Resource mechanism is used heavily within DAWS, not just by the Display objects. All drawing and device management operations are performed using Resources.

The Resource mechanism is the key to DAWS' extensibility.

Returning to the example which led to trouble with inheritance, a display with a mouse and a keyboard would provide mouse server and keyboard server Resources, and at least one drawing Resource. If the display had no keyboard clients, then the keyboard server Resource would never have been activated, so migrating to a display with no keyboard would be trivial. If there *were* one or more keyboard clients, then the keyboard-less display would attempt to find a ResourceFactory to create a remote keyboard server Resource. Such a Resource could be provided either by a stand-alone network keyboard, or by a keyboard 'borrowed' from a nearby workstation. If no suitable remote keyboards could be found, then the original display would be unable to migrate.

3.5.6 Version control

In any complex system such as DAWS, the issue of version control is important, as parts of the system will, over time, be modified to add new features and fix bugs.

There are two main areas of concern to DAWS—implementation changes and interface changes. When an implementation changes, provided the semantics of its interface remain the same, clients of that implementation need not be aware of the modification. In this case, no version control is necessary at a framework level. If the semantics change, the DAWS framework requires that the interface specification is changed, so the issues are then the same as if the interface changes for any other reason.

The CORBA specification[OMG98a] has very little to say about the subject of interface version control[14]. The repository identifier associated with an interface includes a version number consisting of major and minor numbers[15]. If no other version number is specified, version 1.0 is assumed. IDL has a 'version' pragma directive which allows a different version to be specified. For example, the IDL:

```
interface Example {
  ...
};
#pragma version Example 2.3
```

would result in an interface with repository a id. of '`IDL:Example:2.3`'. It was probably the designers' intention that interfaces differing in minor version number would be upwardly-compatible with each other but, in practice, interfaces with any differences in their version numbers are considered to be different. Extensions to existing interfaces are easily performed by declaring a new interface which inherits from the original interface. Incompatible changes to an interface could be represented by a change in its

[14]The yet-to-be-accepted CORBA Meta Object Facility[OMG97] could be used to perform complex version control.

[15]This is only true for the 'OMG IDL' repository id. format which is used by omniORB and many other ORBs. Other formats are permitted.

version number, but that would quite possibly lead to confusion. It is better to declare a distinct interface name[HR93].

DAWS imposes the rule that, once an interface is in use, its definition must not be changed in any way. Extensions to an interface should be performed with inheritance; incompatible changes must be made by declaring a new interface with a different name.

3.5.7 Object relationship representation

One of the requirements outlined in section 3.1 was that DAWS should present a simple management interface to *object location controllers*—systems such as SPIRIT which can make intelligent decisions about object migrations. To make such decisions, the system must be able to find the object references of all of the objects involved in a session, and have some concept of the relationships between objects. The objects within a session form a graph, so some mechanism for storing and maintaining the graph is required. There are a number of possible methods of implementing a suitable service:

1. Register all objects involved in a session with a single session object.

   This would be very simple to implement, but would limit the flexibility and scalability of the system, as notifications of all object creations, migrations, and deletions would have to be sent to a single location. The session object could, of course, be allowed to migrate, but that would require the transfer of a large quantity of state, and would only partly alleviate the problem—sessions would commonly involve objects residing within more than one island of connectivity.

2. Use the facilities of the CORBA Relationship service.

   The Relationship service is an extensive service which permits relationships between CORBA objects to be represented in a high-level manner. Relationships between objects are themselves represented as CORBA objects, so it would be necessary to migrate relationship objects to new locations as the objects they referred to migrated. The Relationship service, like many of the CORBA services, is very generic, and provides far more facilities than would be required by an object location controller. A partial implementation of the service could be used, however.

3. Create a special-purpose distributed session registration service which dynamically adjusted itself to match the distribution of the session's objects.

   To make this method more useful than an implementation of the Relationship service, it would have to be closely linked to the services provided by the object location controller. It would be useful to support a number of different location controller types, so this scheme would be inflexible.

4. Force all interfaces to be derived from a generic relationship management interface (or add relationship management to the `LCObject` interface).

   This would avoid the need to provide a separate distributed object relationship service, but would unnecessarily complicate all object implementations.

5. Represent relationships in an ad-hoc, object-dependent manner.

   This scheme requires that all objects which have relationships with other objects provide some mechanism within their interfaces to find those relationships. Objects are free to use a representation convenient to themselves, placing more of a burden on the migration controller. However, since the controller must have detailed knowledge of the behaviour of different types of object if it is to make intelligent decisions about where they should be located, it is reasonable to expect it to have knowledge of each object's relationship interface.

The final mechanism was chosen for its simplicity.

Any DAWS object which has relationships to other objects must provide a means, within its interface, to find those relationships.

3.5.8 State and caching

One useful aspect of systems like VNC is that the display end-point is stateless. This means that if the display unexpectedly exits, no application information is lost. Although some objects within DAWS are stateless, the majority of them do have some state, so this kind of fault-tolerance is not automatically available.

There are, however, some steps which may be taken to approach the same sort of functionality. Firstly, in some objects, all of the state may be available elsewhere, albeit possibly in a less convenient form. A `Layer` object, for example, could be reconstructed from the data held by the LayerMgr and the Components contained within it.

Another point is that the original reason for splitting the window system into many distributed objects was to gain extra performance when the user is away from their home island of connectivity. This means that what is actually required away from the home base is a *cache* of the home base's state. Objects with a large amount of state should be designed to regularly communicate their state back to their home location. A large text entry box, for example, should send its state back to a stable location approximately every minute. This way, the user would never lose more than about a minute's work.

It will generally be the case that less trust should be placed in machines and object implementations away from the user's home base. This means that the object location controller may choose not to migrate a particular object away from its home location for reasons of stability, even though migrating it would result in increased performance. It is an unproven conjecture of this work that it is possible to strike a proper balance between performance and robustness.

3.5.9 Library and macro support

The DAWS framework provides a library, named DAWSUtil, which has a number of support functions. These include a function to create a location string for a process, and a function to find a reference to the default FactoryFinder object from the Naming service or a file. It also contains the implementation of a class named ResourceHelper which maintains a list of all local and remote Resources currently in use at a ResourceFinder. The class provides all the facilities required for finding remote Resources and controlling Resource object migration.

The framework also provides a C++ pre-processor macro which greatly simplifies the use of the FactoryFinder. The FactoryFinder's FindFactories() operation returns a prioritised sequence of factory object references, some of which may no longer be valid. Finding a valid factory and creating an object with it therefore requires a loop to iterate over the sequence as shown in the code fragment in figure 3.8. When the macro is used, the code becomes much simpler, as shown in figure 3.9.

3.6 Summary

This chapter has introduced the facilities required by DAWS, and has outlined the architecture which has been designed to meet those requirements. DAWS is built upon a general-purpose, distributed, object-oriented framework consisting of:

- A policy to use specific-purpose interfaces wherever appropriate.
- A system of FactoryFinders which permit object factories to be found.
- An LCObject interface which is supported by all objects which may migrate, and the policy that objects are responsible for transferring their own state during object migration.
- A novel system of Resources, ResourceFinders, and ResourceFactories which allows dynamic extension of object functionality, and migration between devices with different basic capabilities.
- A set of library functions and macros which simplify the implementations of objects and clients which use these services.

The next chapter gives details of the design of DAWS itself.

```
DAWSLifeCycle::FactoryFinder_var  ff;
DAWSLifeCycle::Locations          locs(1);
DAWSLifeCycle::FactorySpecs_var   fspecs;
WindowFactory_var                 wf;
Window_var                        win;

ff     = DAWSUtil::GetFactoryFinder(orb);
locs[0] = (const char *)"ALL";
fspecs = ff->FindFactories(locs, Window_IntfRepoID);

int bad = 1;
for (unsigned int i=0; i < fspecs->length(); i++) {
  try {
    wf  = WindowFactory::_narrow(fspecs[i].f);
    win = wf->CreateWindow(w, h);
    bad = 0;
    break;
  }
  catch(...) {
  }
}
if (bad)
  throw DAWSLifeCycle::NoFactory();
```

Figure 3.8: Finding a factory and creating an object without macro support

```
DAWSLifeCycle::FactoryFinder_var  ff;
DAWSLifeCycle::Locations          locs(1);
Window_var                        win;

ff     = DAWSUtil::GetFactoryFinder(orb);
locs[0] = (const char *)"ALL";

DAWSFACTORY(win, ff, locs, Window_IntfRepoID,
            WindowFactory, CreateWindow(w, h));
```

Figure 3.9: Finding a factory and creating an object with macro support

4
Window System Design

This chapter presents the design of DAWS itself. It first gives a detailed example showing the Resources used in a running system. Next, some of the basic design decisions about pixel representations and input formats are outlined. Finally, the interface designs of the most important objects in the system are presented. More details regarding object implementations, and the designs of less important objects, may be found in Chapter 6.

There are three basic principles which drove the design:

- Low latency, to make the system as fast as possible, despite network delays.
- Platform flexibility, to be portable to a very wide range of display and input devices.
- Adaptability and extensibility, to cope with changing circumstances at run-time, and to extend to new devices.

4.1 An example configuration

Figure 3.1 showed an overview of just the top-level objects used by DAWS. Much of the window system functionality is actually carried out by Resource objects 'within' these top-level objects. Figure 4.1 shows a more detailed example of the low-level parts of a running DAWS session, including the Resources involved. The reasons for this design are explained in the following sections.

There are two major Resource types—ones which perform rendering operations, and ones which provide access to input events. The figure shows a `Display` object with two of each kind: the `Mouse::Server` and `Keyboard::Server` allow clients to register an interest in mouse and keyboard activity; the `Draw24::Draw` provides simple pixel-based rendering operations with 24 bits-per-pixel; the `Draw24::DrawShape` provides operations which render into non-rectangular regions.

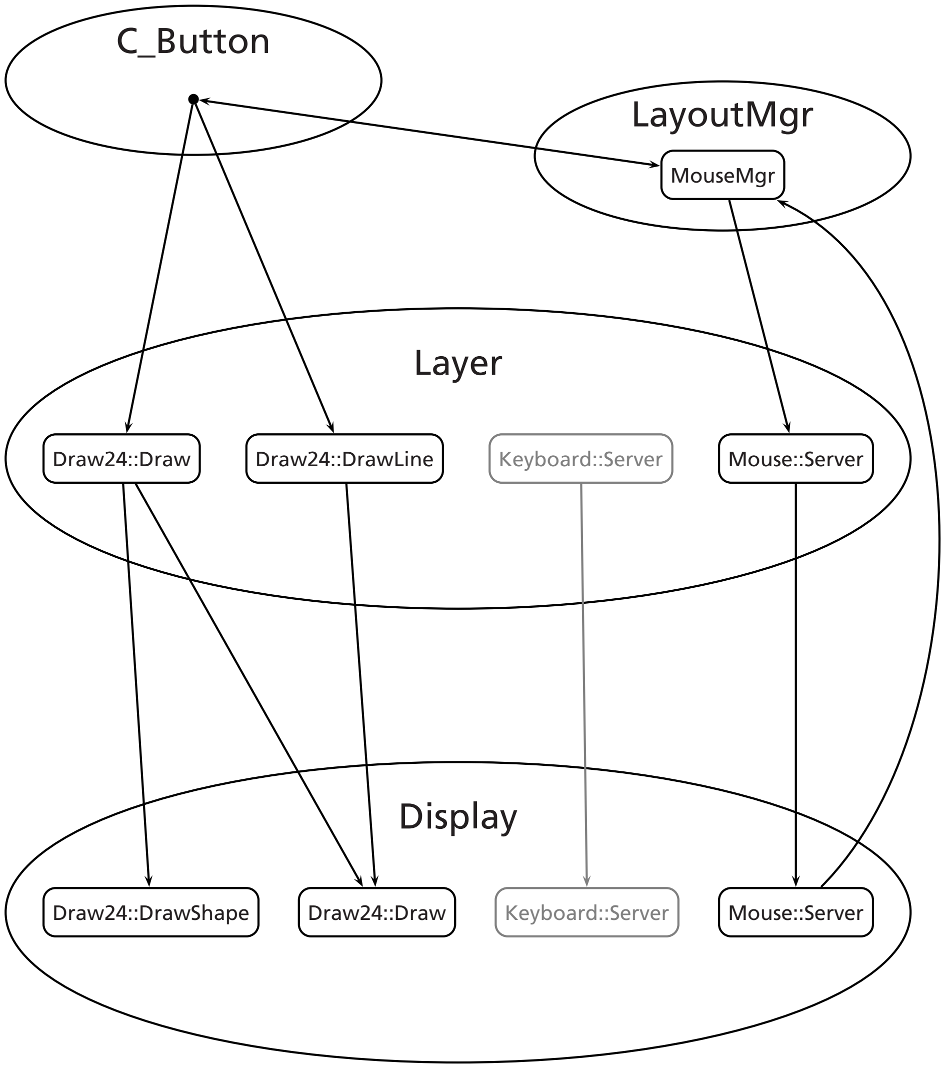


Figure 4.1: Some of the Resources involved in a running DAWS session

The Layer object provides a clipped region within the display. The Mouse::Server and Keyboard::Server Resources provide managed access to the Display's input servers, so that Components within the Layer need not know the Layer's current position, and whether it is activated or not. As explained later in section 4.3, the diagram shows that the input events themselves are sent directly from the Display's Resources to the objects which require them, rather than passing through the Layer's Resources.

The Layer also has two Resources which provide clipped rendering operations. The Draw24::Draw provides clipped versions of the operations provided by the same Resource within the Display. One of the drawing operations renders a rectangular block of pixels. When such a block is clipped, it may result in an arbitrary shape. If the Display has a Draw24::DrawShape Resource (which the one in the figure does), the Layer uses it to render a clipped rectangle. If the Display only supports the Draw24::Draw Resource, the Layer performs multiple rectangle drawing operations.

The Draw24::DrawLine Resource provides operations to draw simple straight lines. This particular Display has no comparable Resource, so lines must be drawn as multiple points via an operation in Draw24::Draw. Other Displays could support line drawing. The facilities of the different drawing Resources are explained in section 4.4.

The C_Button object is a GUI Component (as are all objects with interface names beginning 'C_') which provides a simple push-button with an image on it. It uses the Layer's drawing Resources to perform its rendering. Rather than using the Layer's input servers directly, the Component uses a MouseMgr Resource provided by the LayoutMgr. The MouseMgr provides a simplified interface to the most common Component requirements. It uses the LayoutMgr's knowledge of the Component layout to direct the mouse events to the Component currently under the mouse pointer, and perform extra event filtering. Components are free to communicate directly with the Layer's input servers if they have more complex input requirements.

In the example, there are no clients requiring keyboard input at all. This means that the Keyboard::Server Resources in the Layer and Display need not actually be loaded (and are hence shown in grey). The first time a client requests a Keyboard::Server, the implementation objects are created[1].

4.2 Imaging model

There are two schools of thought over how imagery should be specified at the lowest levels of a windowing system: some believe that all rendering operations should be resolution independent, others that they should be resolution dependent. Resolution independent systems usually use floating-point representations of coordinates whereas resolution dependent ones use integer coordinates, with a one-to-one mapping between integer values and pixels.

DAWS takes the resolution dependent route for two reasons. Firstly, it is a basic requirement that it should be possible to perform precise image rendering, which is inconvenient with floating-point coordinates. Secondly, some of the thin display devices which should support DAWS have no hardware floating-point units, so manipulating floating-point coordinates would be a significant processing burden for them.

[1]Removing a Resource object once it has been created is more troublesome, since CORBA does not track clients holding object references. The implementation object *could* safely be deleted, as long as the CORBA system was able to re-start it on demand.

DAWS uses signed 16-bit integers (IDL shorts) to represent coordinates. 16-bit values are used as a trade-off between transmission size and resolution support. Current displays have a resolution of at most a few thousand pixels across, so a move to 32-bit values would not be useful.

There are, of course, advantages to using a resolution independent imaging model at higher levels in the window system. This is especially true for applications which provide WYSIWYG document editing. DAWS has been designed to support resolution independent drawing Resources by providing an interface to information about the physical screen resolution. This kind of Resource would be added to the Layers (or be provided by stand-alone ResourceFactories), and some displays. Other displays would still only support resolution dependent imaging, requiring translation by the Layer's Resources. Resolution independent drawing Resources have not been designed for the prototype version of DAWS, since useful support would require a large quantity of work which would not be relevant to the central issue of this dissertation. A production version of DAWS *would* come with resolution independent rendering support.

4.2.1 Colour

Colour support can be another complex issue, particularly if it is to be integrated with colour printing. DAWS currently only supports a naïve representation in which all colours are specified as red-green-blue triples with 8 bits per colour Component (i.e. 24 bits-per-pixel). This is sufficient for a research system, but a production version should also support named colours, and different low-level display capabilities, such as monochrome and palette-mapped devices, as well as colour correction between different devices.

This would fit neatly into the DAWS architecture: each Display object would support Resources providing one or more different low-level colour schemes. Higher level objects, such as the Layer, would support drawing Resources providing symbolic colour representations.

4.3 Input and events

DAWS splits its support for input events into two kinds—low-level basic events, and high-level symbolic events. Low-level events are ones such as mouse movements and key presses which come directly from some hardware device; high-level events are ones representing semantic occurrences such as the press of a button Component, or the selection of a menu item.

High-level events are specified in a generic manner, using the CORBA any type, in a similar manner to the standard CORBA Event service. The Event service provides a standard means for *suppliers* to send events to *consumers*, possibly via an intermediary *event channel* object. Consumers may receive events via either *pull* or *push* mechanisms. With the pull mechanism, the consumer invokes operations on the event channel to check for and receive events; with push, the consumer is a CORBA object with an operation which is called by the event channel when an event occurs.

DAWS uses a simplified scheme which uses only the push model. In addition, the events consist of a text string key along with an `any` value, rather than just an `any`. This design fits better with the requirements of GUI Components, as explained later in section 4.7.3.

Low-level events, especially ones providing continuous updates like mice, are far more dependent on latency than the higher-level ones. It is also useful, from a programming point-of-view, for objects receiving such input events to be able to distinguish different types of event at an operation level, rather than having to programatically test for particular events. For these reasons, low-level events are passed through specific-purpose interfaces as appropriate operation invocations. The interfaces currently available provide access to mouse and keyboard events; interfaces to others sorts of input devices, such as joysticks and speech input, could obviously be defined.

> There is nothing special about mouse and keyboard input in DAWS; other forms of input can be supported using exactly the same mechanisms.

4.3.1 Mouse input

The main purpose of the DAWS mouse interfaces is to pass information about mouse events from the display (or a stand-alone mouse) to Components within different Layers. Components should not have to be aware of their Layer's current location to understand the mouse coordinates. The simplest way to achieve this would be to send mouse events from the display to each Layer, and then from the Layers to the Components. This would permit the Layer to adjust the coordinates of each event.

This scheme is not used since it would add an extra stage to the event path, and thus add latency. Instead, the `Mouse::Server` Resource in the display maintains a record of the offset of each of its clients, and adjusts the events it sends accordingly. Each Layer has its own `Mouse::Server` Resource to which the Components connect. When a Component registers with the Layer's mouse server, the Layer registers the Component with the display's mouse server. The Layer then keeps the display up-to-date with its current location so that the events are properly translated. The events are thus sent directly from the display to the Components, with translation controlled by the Layers.

Figure 4.2 shows the format in which mouse data is transmitted, and the interface for clients which receive mouse events. The coordinates are absolute, rather than rela-

```
module Mouse {
  typedef unsigned long Buttons;
  const unsigned long LMB = 1;   // Left button
  const unsigned long RMB = 2;   // Right button
  const unsigned long MMB = 4;   // Middle button

  struct Data {
    Drawing::Coord   X;          // X coordinate
    Drawing::Coord   Y;          // Y coordinate
    Buttons          btns;       // Buttons. 1 for pressed.
    DAWSTime::Stamp  time;       // Timestamp.
  };

  interface Client {
    oneway void MouseHint(in Data dta);
    void MouseData(in Data dta);
    void MousePause(in Data dta);
  };
};
```

Figure 4.2: Mouse data format and client interface

tive to the most recent position. The button state is represented with a 32 bit integer in which each bit corresponds to a particular button on the mouse. On a normal three-button mouse, the left button is bit 0, the right is bit 1, and the middle is bit 2. The timestamp is used to detect double-clicks.

Mouse clients may request a range of different kinds of events to be pushed to them by the mouse server. If they request complete position information, they receive an event (i.e. an operation invocation) every time the mouse position or the button state changes at all. Alternatively, they may provide three different button masks. The first mask requests an event whenever one of the specified buttons is pressed; the second requests an event when one of the buttons is released; the third requests full position updates as long as one of the specified buttons is held down. Finally, clients may also request a special event if the mouse has paused in a single location for a while. This is useful for implementing tool-tips—small help messages for controls.

The client interface has three operations which are called by the server. The MouseHint() operation is used to send 'unimportant' updates which do not involve a change in button state. It does not matter if such events are lost, so the operation is declared as being oneway, gaining a measure of asynchronous behaviour. MouseData() is used for updates which *do* involve a change in button state. It cannot be declared oneway since it is important that button clicks and releases are registered. Finally, MousePause() is used to indicate that the mouse has paused.

Note that all mouse client objects must support all three operations, even if they know that not all of them will be called—a client which has only requested button click information will only ever receive calls to its MouseData() operation. In this sort of situation, the client implementation can have empty implementations of the unused operations.

The mouse server interface is relatively uninteresting, consisting mainly of operations to add and remove clients, activate and deactivate them, and modify the events and position offsets they require. The server interface also provides GrabServer() and UnGrabServer() operations which allow single clients to request exclusive access to the mouse data. This is useful in various situations. For example, when the user is moving a vertical scroll-bar, they should be able to stray horizontally away from the scroll-bar while they have the mouse button held down, without mouse events being sent to other Components the pointer passes over. These operations differ from the equivalent functions in X (see page 11), in that the grab only lasts until the button state changes, at which time the client must re-request the grab if it still requires it. It is also possible for another client to 'steal' the grab. These two provisions avoid the problem in X where a client can grab input and then crash, leaving the X server in an unusable state.

4.3.2 Keyboard input

Keyboard input is broadly structured along the same lines as mouse input. Once again, the input events are sent directly from the display's Resource to the Components, with a second Resource in the Layer which activates and deactivates keyboard clients as necessary. Unlike mouse data, there is no event translation necessary, so the Resource in the Layer is only used to manage client activation.

There are actually two different keyboard client interfaces. One provides 'raw' data which includes full details of the modifier keys (such as shift, control, …), whether the key was pressed or released, and a timestamp. The second interface simply permits the transmission of a single Unicode[Uni96] character. This second interface is sufficient for the vast majority of clients.

The raw format is based upon the *keysym* definitions used by X. Each key which may exist on a keyboard has an associated 16-bit value. The value is transmitted as a 32-bit unsigned integer. The low 16 bits contain the key value, bits 16–24 represent modifier keys, and bit 31 indicates whether they key was pressed or released; bits 25–30 are currently unused.

The key value depends upon what is printed on the key, not upon where the key is located on the keyboard. DAWS does not provide an interface to information about where keys are located on the keyboard, as it is very difficult to provide a completely implementation-independent representation. The keyboard mapping in X, for example, is quite awkward, and does not extend to esoteric keyboard variants such as chording keyboards[CTD98]. In DAWS, keyboard mappings are implementation dependent; all display (and stand-alone keyboard) implementations should provide an interface which permits the user to re-define the meanings of the available keys.

A limited set of applications, notably games, may wish to know the physical locations of keys on the keyboard, so an interface providing the equivalent of X's *keycodes* could be provided as a separate Resource. This would still have to make more assumptions about the types of keyboard device in use than DAWS currently does.

Like the mouse server, the keyboard server permits a single client to 'grab' all input events for a while. One of the primary purposes of this is to prevent other clients intercepting the characters as the user types a password. It is therefore not appropriate for other clients to be able to 'steal' a grab[2]. To avoid the X grabbing problems, a client performing a grab must nominate at least one key, usually the 'Escape' key, which will release the grab automatically.

Both of the keyboard event formats are highly inefficient with network traffic, since each event, requiring a network round-trip, only contains information about a single key press (or release in the case of raw events). Higher level events within the system, such as communication between Components, batch key presses into strings.

4.3.3 Ultra-thin devices

Both the mouse and keyboard servers presented here are reasonably simple to implement, but they do require a certain amount of processing, particularly because they must cope with multiple clients with differing requirements. If the interfaces were considered too complex for a very simple input device to support, new lower-level input interfaces, supporting a single client, could be defined. A ResourceFactory running at a location close to the input device would then be used to provide clients with the interfaces described above.

4.4 Display and drawing

As well as input, it is obviously vital for a window system to provide a flexible display mechanism, and a powerful set of rendering functions.

The DAWS `Display` object has two primary purposes. The main display activities of handling rendering and input are carried out by Resources, so the `Display` is derived from the `ResourceFinder` interface. Other objects within the system must be able to find out about the physical attributes of the display, such as its colour model and its resolution. Such information is provided by attributes of the `Display` object. 'Teleporting' the user's session from one display to another is performed by migrating the `Display`

[2]With the current version of DAWS, this is somewhat redundant since clients are not authenticated in any way. They may thus easily masquerade as the client holding the grab, in order to release it. Use of the Security service would prevent such attacks.

object. This appears to clients as if the display's physical attributes have changed, so clients may register with the Display to receive updates when the display information changes.

When it comes to rendering functions, DAWS' Resource mechanism provides for a great deal of flexibility, in that new rendering functions may be added in a simple manner. If new drawing Resources are defined, existing display implementations will become out-of-date, because clients designed to use the new Resources will not be able to use older displays' older Resource types. This situation can be avoided by providing ResourceFactories which create stand-alone versions of the new Resources to 'wrap' around the old Resources of the old displays. Similarly, if new displays only support new Resources, old clients will not be able to use them, so ResourceFactories should be used to provide wrappers around the new Resources.

Unlike some systems, DAWS has an asynchronous drawing model, in which clients may perform drawing operations at any time, rather than having to request a re-paint from the system. This not only makes the programming of painting functions much simpler, but also reduces latency—requesting a re-paint would involve at least one extra network round-trip. This is possible since all drawing operations are clipped as they pass through the Layers.

DAWS currently has a set of drawing interfaces which provide relatively low-level rendering operations. These interfaces could provide the basis for higher-level interfaces, but they are, in themselves, quite sufficient to build complex GUIs. As mentioned earlier, DAWS currently only supports true-colour devices, with 24 bits-per-pixel[3]. Figure 4.3 shows the IDL definitions used by the base drawing interface.

The Drawing module contains colour model independent definitions, the most important of which are shown in the figure. The Draw24 module contains a number of different drawing Resources, the most basic of which is Draw24::Draw. The Draw interface is a somewhat arbitrary trade-off between simplicity, to permit very simple displays, and functionality, to build useful systems. Note that throughout DAWS, the coordinates (0,0) represent the top-left corner of the display.

The simplest drawing operation, plotting a single point, is not very useful at all. If more than one point is required, it is far more efficient to specify multiple points in one operation invocation, so as to avoid network latency. (Some performance figures for the prototype system may be found in section 7.1.) There are several ways in which multiple points may be specified: MPoint() and CPoint() plot points in arbitrary locations, in a single colour, and different colours respectively; Block() draws a rectangular block of solid colour; RMap() renders a rectangular 'map' of different coloured pixels. MPoint() and CPoint() are useful for drawing lines and curves, Block() is mainly used to

[3]It is, of course, possible for a display to reduce the number of significant bits to support hardware with lesser capabilities. The XDisplay described in section 6.1 supports 8, 15, and 16 bits-per-pixel displays in this manner.

```
module Drawing {
  typedef short Coord;            // A coordinate
  struct Point {                  // A point
    Coord X;
    Coord Y;
  };
  typedef sequence <Point> Points;
};

module Draw24 {
  struct PointPixel {
    Drawing::Coord X;
    Drawing::Coord Y;
    octet          R;
    octet          G;
    octet          B;
  };
  typedef sequence <PointPixel> PointPixels;

  // An array of 24-bit pixels, 3 bytes per pixel in RGB order:
  typedef sequence <octet> Array;

  struct RectMap {                // Rectangular map of 24-bit pixels
    Drawing::Coord X;             //  X Drawing::Coord
    Drawing::Coord Y;             //  Y coord
    Drawing::Coord W;             //  Width; height from data
    Array          data;          //  The data stored row-by-row
  };

  interface Draw : DAWSLifeCycle::Resource {
    void Point  (in Drawing::Coord x, in Drawing::Coord y,
                 in octet r, in octet g, in octet b);
    void MPoint(in Drawing::Points p,
                 in octet r, in octet g, in octet b);
    void CPoint(in PointPixels p);
    void Block  (in Drawing::Coord x, in Drawing::Coord y,
                 in Drawing::Coord w, in Drawing::Coord h,
                 in octet r, in octet g, in octet b);
    void RMap   (in RectMap r);
  };
};
```

Figure 4.3: IDL definitions for the base drawing interface

clear large regions of the display, and RMap() is used to render complex images.

The RectMap structure, used by RMap() (and other operations), has an interesting definition. Although the drawing function has to know both the width and height of the map in order to render it, it is only necessary to explicitly send the map width since the height may be derived from the length of the data sequence. The colour data itself is represented as a sequence of octets with three octets per colour value. An alternative, semantically richer, possibility would be:

```
struct Colour {
  octet R;
  octet G;
  octet B;
};
typedef sequence <Colour> ColourArray;
```

Unfortunately, this approach leads to a severe marshalling overhead when the data is transmitted. Sequences of octets are 'special', in that it is guaranteed to be safe to transmit an octet sequence across the network without any translation; other types, especially constructed types, are likely to have representations in memory that differ from their marshalled form. Marshalling a sequence of structs therefore involves unpicking each item in the sequence. For this reason, DAWS sacrifices a little clarity for a large increase in performance.

4.4.1 Supplementary drawing Resources

There are a number of other Resources defined within the Draw24 module. The DrawCBlock interface provides a single operation, CBlock(), which draws multiple rectangular blocks with different colours. This is a sufficiently simple, and useful, operation that it should probably have been a part of the main Draw interface. It happened that the need for such an operation did not become apparent until quite high-level parts of the system were being developed, a significant time after the original Draw interface was designed. If a new operation had been added to Draw, many parts of the system would have had to be modified, or at least re-compiled. Adding a new Resource meant that existing drawing implementations and clients continued to work with no modifications. The DrawCBlock interface is thus an excellent, genuine, example of the extensibility of DAWS' Resource mechanism.

The remaining Resources provide more advanced rendering operations. It is often useful to render images with non-rectangular outlines, and DAWS provides three different Resources which permit this. The DrawShape Resource allows the rendering of solid shapes, and coloured maps, based upon the Shape representation used for clipping, explained in section 4.5. DrawMask allows a bit-mask to be specified, in which each pixel is either active or not; it provides operations to render the active pixels in

```
interface BlockManip : DAWSLifeCycle::Resource {

  void BlockCopy(in Clipping::Shape shpe,
                 in Drawing::Coord  toX, in Drawing::Coord toY);

  typedef long BlockNum;

  exception CannotSave  {};
  exception NoSuchBlock {};

  BlockNum BlockSave(in Clipping::Shape shpe) raises (CannotSave);

  void BlockLoad(in BlockNum       blk,
                 in Drawing::Coord toX,
                 in Drawing::Coord toY,
                 in boolean        free) raises (NoSuchBlock);

  void BlockFree(in BlockNum blk) raises (NoSuchBlock);
};
```

Figure 4.4: IDL definitions for the block manipulation Resource

a solid colour, or from a map. Finally, DrawAlpha uses an alpha map to blend either a solid colour, or a colour map, with the existing background imagery. DrawAlpha is particularly useful for rendering anti-aliased text over arbitrary imagery (see section 4.8.3).

The final Resource in Draw24 is DrawLine which provides operations to render simple lines and paths.

4.4.2 Block manipulation

The drawing operations discussed so far, with the slight exception of alpha channel rendering, are only concerned with drawing new images provided by the client, independent of the current contents of the display. Experience with VNC has shown that adding operations which copy pixels from one part of the display to another results in a significant increase in performance. DAWS builds upon VNC's single 'copy rectangle' operation by providing the BlockManip Resource, shown in figure 4.4. Note that BlockManip is not part of Draw24 since its operations are independent of the colour model in use.

The BlockCopy() operation copies an arbitrarily shaped block of pixels from one location to another. (The shape representation is described below in section 4.5.) This operation is especially useful for implementing such user interface features as image scrolling and solid window dragging.

The Resource may optionally permit clients to save blocks of pixels to be rendered later. If the Resource does not support block saving at all, or has insufficient memory to save the requested block, it throws the `CannotSave` exception. Block saving is useful for temporarily saving the background when a transient item such as a menu is displayed, and also for storing often-used images, such as desktop background patterns.

Saved blocks are freed either by an explicit call to the `BlockFree()` operation, or by setting the `free` argument to `true` in a call to `BlockLoad()`. If these were the only means to free blocks, an errant client could consume the Resource's memory, and never free it. The Resource is therefore permitted to free a block at any time. This means that clients of the Resource must always be able to fall-back on some other means of rendering if they receive a `NoSuchBlock` exception, and also that the Resource must be careful about re-using block numbers.

4.5 Shape representation and clipping

Several previous sections have mentioned the DAWS shape representation, without giving any details of it. All window systems which provide overlapping windows must have some means of representing the shapes of regions, in order to perform clipping of graphics. The most common shape representation is a list of rectangles whose union is the required shape. This is an especially obvious format for those window systems which provide only rectangular windows since, in such systems, all non-rectangular regions are formed by occluding part of one rectangle by parts of other rectangles.

The problem with using an unsorted list of rectangles is that operations involving regions are very inefficient. If n is the number of rectangles in a region, for example, then finding if a point lies inside the region involves testing an average $n/2$ rectangles; finding if a point is outside the region always involves testing all n rectangles. Even worse, operations which combine two regions are $O(nm)$, where n and m are the numbers of rectangles in the two regions. Another difficulty is that when two shapes are combined in some manner, some of the resulting rectangles may fit together to form larger rectangles. The system must spot such situations, otherwise the number of rectangles contained in a shape would tend to increase every time an operation was performed.

As with many similar problems, the solution is to sort the rectangles in some manner. Unfortunately, there is no obvious way to sort a set of arbitrary rectangles, since the full extent of each rectangle must be taken into account. Instead, DAWS uses a representation based upon the scheme described by Foley et al.[FvDFH90], in which regions are broken up into vertical and horizontal *spans*. Figure 4.5 shows the IDL definitions which DAWS uses. An example shape is shown in figure 4.6.

The way in which shapes are broken into spans is similar to the rise-and-run-length encoding used by VNC, described in section 2.5.2, used in monochrome. The shape is

```
module Clipping {
  struct XSpan {
    Drawing::Coord from;          // Inclusive
    Drawing::Coord to;            // Exclusive -- span does not
  };                              //  include this point
  typedef sequence <XSpan> XSpans;

  struct YSpan {
    Drawing::Coord from;          // Inclusive
    Drawing::Coord to;            // Exclusive
    XSpans         xspns;
  };
  typedef sequence <YSpan> YSpans;

  struct Shape {                  // Bounding box:
    Drawing::Coord X;             //  Top-left X coord.
    Drawing::Coord Y;             //  Top-left Y
    Drawing::Coord W;             //  Width
    Drawing::Coord H;             //  Height
    YSpans         yspns;         // Spans
  };
};
```

Figure 4.5: IDL definitions for the DAWS shape representation

sliced into vertical (Y) spans which are, in turn, sliced into horizontal (X) spans. Each span indicates a range which falls within the shape. The spans are stored in numerical order. In figure 4.6, the shape is split into five Y spans, each of which has either one or two X spans within it. Note that all Y spans must have at least one X span in them—if there is a gap in the shape, no Y span is stored for that area.

The advantage with using the span mechanism is that, since the spans are stored in order, operations which were inefficient with a simple list of rectangles become much more efficient. Finding whether a point lies within the shape requires a binary search through the Y spans to see if the Y coordinate of the point is contained in a Y span. If it is not, the point is outside the shape; if it is, a binary search through the X spans is performed. The operation is therefore $O(\log n)$, where n is the number of spans. Although there will, in general, be slightly more spans than rectangles in a shape, this is a significant improvement, especially for complex shapes.

Combining shapes with each other is also much more efficient, since the operation can be performed with a single linear pass through the two shapes. Because of the constraints placed on the spans, it is simple to merge neighbouring spans to form larger spans. The details of the shape manipulation functions used by DAWS are presented in section 6.2.1.

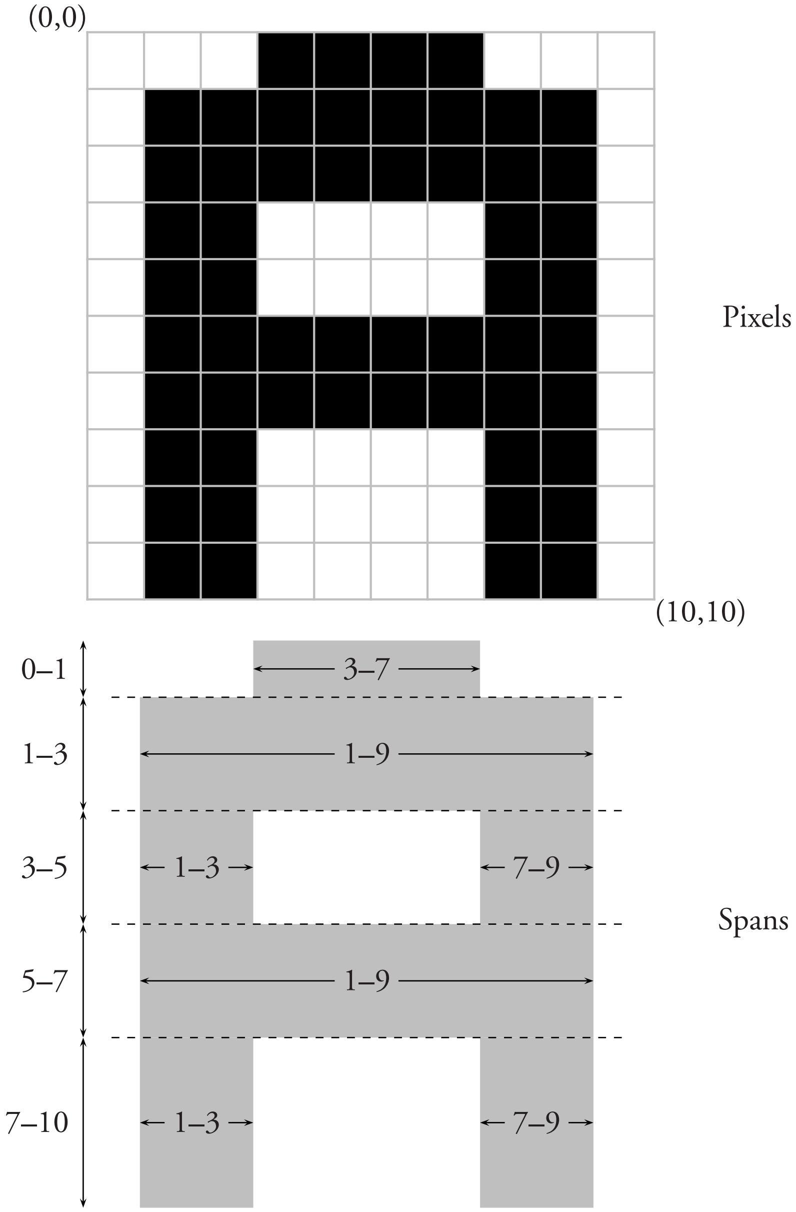


Figure 4.6: The DAWS shape representation

4.6 Layers and Layer management

One of the primary uses of the shape mechanism is in representing overlapping regions within the window system, so that drawing operations may be clipped. Each shaped region has a `Layer` object which keeps a record of the region's 'natural' shape, and currently visible shape. The Layer's drawing Resources are responsible for clipping the rendering operations, based upon the currently visible shape.

Unlike many window systems, Layer objects in DAWS are not inherently capable of containing other Layer objects, although it would have been reasonably simple to permit such behaviour. DAWS discourages this recursive approach, since, in a distributed system, it is best to keep the system as 'flat' as possible. To use nested Layer objects would greatly increase the number of potentially remote operation invocations that were required to render a desktop, thus increasing latency. DAWS is, however, sufficiently flexible that it would be simple to implement a user interface Component (see section 4.7) which was derived from the Display interface, permitting a Layer to recursively contain more Layers.

The Layer objects themselves do not know about any of the other Layers in the system. Each session has a single LayerMgr object which records the natural shapes of each of the Layers, their locations, and their ordering. The LayerMgr is responsible for combining the natural shapes of the Layers to find their visible shapes, and communicating this information to the Layer objects. All requests for changes to a Layer's location, size, or depth must therefore be made to the LayerMgr, rather than directly to the Layer object in question.

The consequence of this object arrangement is that many of the Layer interface's operations may only be called by the LayerMgr, whilst others may be called by any client. DAWS could 'hide' the management operations in another interface derived from the main Layer interface, to prevent clients from seeing the operations they were not permitted to use. However, CORBA provides no means of enforcing a restriction such as this—it would not be possible to prevent a malicious client from narrowing a reference to a Layer into a reference to the derived, 'private', interface, and calling the restricted operations. Another reason for not using inheritance in this way is that it is useful to be able to extend the functionality of the Layer interface, as is done with the LayoutLayer, mentioned in section 4.7. DAWS therefore declares the restricted operations in the main Layer interface, and relies on trusting the Layer's clients. It *would* be possible to enforce the necessary restrictions with an implementation of the Security service.

There are a number of subtleties to the way in which changes in Layer arrangement are handled. The most obvious is that, since it is not possible to update the clipping shapes of all Layers simultaneously, Layer drawing operations must be 'suspended' while updates are in progress. In a distributed environment, updating the visible shapes of all Layers may take a considerable time, due to network latency. DAWS attempts to alleviate this to some extent, by assuming that the Layers nearest the 'top' of the stack of Layers are the ones of greatest interest. When the LayerMgr is re-calculating the visible portions of its Layers, it therefore suspends the output of each Layer in turn, starting with the lowest, and then re-enables the Layers with their new shapes, starting with the highest.

Another complication with Layer management arises when Layers are re-sized, due to the fact that DAWS permits Layers to have non-rectangular natural shapes. When a

```
module Size {
  struct Range {
    Drawing::Coord minW;        // Minimum width
    Drawing::Coord minH;        // Minimum height
    Drawing::Coord baseW;       // Base width
    Drawing::Coord baseH;       // Base height
    Drawing::Coord maxW;        // Maximum width
    Drawing::Coord maxH;        // Maximum height
  };
  struct Geometry {
    Drawing::Coord X;           // Top-left X-coordinate
    Drawing::Coord Y;           // Top-left Y-coordinate
    Drawing::Coord W;           // Width
    Drawing::Coord H;           // Height
  };
  union VisibleGeometry switch (boolean) {
    case TRUE: Geometry geom;
  };
  typedef sequence <Size::VisibleGeometry> VisibleGeometries;

  exception OutOfRange {};
};
```

Figure 4.7: IDL structures for size ranges and geometries

Layer is rectangular then, when its size is changed, it is simple for the LayerMgr to calculate its new natural shape. When a Layer has a non-rectangular natural shape, however, the LayerMgr must query the Layer object about its new natural shape before it is able to calculate the Layer's new visible shape.

4.7 User interface Components

The parts of DAWS discussed so far provide all of the essential elements of a window system—graphics may be rendered and clipped into arbitrarily-shaped Layers, and input events may be requested and received. It would be possible to create applications using just these services, but to do so would involve great duplication of effort between different applications. Applications may of course do this if they wish, but most will wish to use the higher-level facilities provided by re-usable user interface *Components*. The layout of Components within a Layer is managed, in a very simple way, by a LayoutMgr object. There is an extension to the Layer interface, named LayoutLayer, which associates a LayoutMgr with a particular Layer.

Components cover a very wide range of functionality, from simple controls such as buttons and scroll-bars, through layout algorithms, to complex GUI elements like file requesters and text editors. All of these are actually elements of the GUI, rather than the window system, and are thus outside the scope of DAWS. The standard `Component` IDL interface, supported by all Components, is, however, an important part of the design of DAWS.

At the lowest level, a Component is simply an object which looks after a shaped region of a Layer, within which some imagery is rendered. The object may also request and process some input events. The imagery may be provided directly by the Component, may be formed from a number of sub-Components, or may be a combination of sub-Components and original imagery. The arrangement of Components within a Layer therefore forms a tree. Some Components create their own sub-Components; others receive references to sub-Component objects when they are initialised.

For the majority of Components, the most efficient method of rendering themselves is simply to draw all of their imagery, regardless of whether it falls within the Layer's current visible shape, relying on the Layer to do the clipping. Some activities, such as motion video display, *do* benefit from being clipped by the Component. Components are thus able to request that the `Layer` object keeps them informed about its current clipping region. This is achieved through the event mechanism described below in section 4.7.3.

Most Components do not have a single fixed size, but support a *range* of sizes. A size range takes the form of a preferred, or *base*, size, a minimum size, and a maximum size. The structures used by DAWS to represent size ranges and geometries are shown in figure 4.7. Components which perform layout of other Components derive their size ranges from the size ranges of the Components they contain. As with Layers, non-rectangular Components must be queried about their shapes at different sizes.

The `Component` interface, shown in figure 4.8 is the only major generic interface in DAWS. Any Component may be initialised by giving it a sequence of arguments consisting of key, value pairs, and a sequence of the Components (if any) it should contain. Individual Component interfaces are permitted to also provide Component-specific initialisation operations if they wish. Components, like all top-level objects in DAWS, are created using object-specific factories which are found through the FactoryFinder. All Component factories must be derived from the `ComponentFactory` interface. Component initialisation is performed as a separate step from object creation, since it is sometimes useful to create circular dependencies between Components. For example, if a numeric input control was formed from a combination of a text-entry box and a scroll-bar, a reference to the scroll-bar object would appear in the text box's arguments, and a reference to the text box would appear in the scroll-bar's arguments, so that the numeric value was automatically passed between the two objects. Such inter-Component communication is performed through the Event mechanism, described in section 4.7.3.

There are two primary reasons for using a generic interface in this part of the sys-

```
interface Component;
typedef sequence <Component> Components;

interface Component : DAWSLifeCycle::LCObject
{
  struct Argument { string key; any val; };
  typedef sequence <Argument> Arguments;

  readonly attribute string                type;
  readonly attribute string                name;
  readonly attribute Size::Range           sizeRange;
  readonly attribute Size::VisibleGeometry visGeom;
  readonly attribute Components            contents;
  readonly attribute Clipping::Shape       shpe;

  exception ArgError     { string key; string message; };
  exception ContentError { string message; };

  Size::Range Initialise (in Arguments args, in Components cont)
    raises (ArgError, ContentError);

  Size::Range ReSize();

  Clipping::Shape CalcShape(in Drawing::Coord w,
                            in Drawing::Coord h)
    raises (Size::OutOfRange);

  void SetGeometry(in Size::VisibleGeometry vgeom,
                   in boolean               suspend)
    raises (Size::OutOfRange);

  void Paint(in Clipping::Shape dirty);
};

interface ComponentFactory
{
  Component CreateComponent(in LayoutMgr lmgr, in string name)
    raises (DWSLifeCycle::CannotCreate);
};
```

Figure 4.8: IDL definitions required by DAWS GUI Components

tem. Firstly, most Components have a large number of optional arguments, used to over-ride the Component's defaults. A simple button with a text label has optional arguments to specify the text font, the text size, and the colours of the text and button, for example. Such options are difficult to specify in IDL without a generic approach, requiring either a separate operation for each argument, or the use of unions discriminated by booleans, like the Size::VisibleGeometry union shown in figure 4.7.

The second, more important, reason is that the generic interface permits applications to take a highly flexible approach to creating their user interfaces. In particular, it is very simple to create and initialise Components based upon a GUI description language, similar to the FormsVBT[BM94] system provided with Modula-3, or from a general-purpose scripting language. Since all Components are created and initialised in the same way, description and scripting languages are able to create any type of Component, not just a limited set chosen by the language designer. The design and implementation of a simple GUI description language is presented in section 6.5.

4.7.1 Component layout

When a Component has been initialised with its arguments and contents, with the Initialise() operation, it calculates its size range, based upon the size ranges, and possibly shapes, of its contents (if it has any). Leaf Components in the Component tree must therefore be initialised before the Components which contain them. Once all of the Components in the tree have been created, the top Component is registered with the LayoutMgr object. At this stage, the Layer (represented by a LayoutLayer object) can be installed in the LayerMgr. The LayerMgr then uses some means (usually a window manager—see section 4.8.2) to set the Layer's geometry. If the Layer is rectangular, the LayerMgr is also able to calculate the Layer's visible shape, and instruct the Layer that it may begin painting; if the Layer has a more complex shape, all of the Component positions must be decided before any painting may occur.

When the Layer receives its geometry from the LayerMgr, it informs its LayoutMgr, which is, in turn, responsible for managing the layout of the Components. The LayoutMgr, however, knows nothing at all about what sort of layout algorithms may be employed by its Components, so it must delegate control to the Components themselves.

DAWS uses the obvious recursive approach: the LayoutMgr calls the top Component's SetGeometry() operation with the geometry of the Layer; the top Component calculates the geometries of its children and calls their SetGeometry() operations, and so on. A Component may be informed that it is not currently visible, by setting the discriminant of the Size::VisibleGeometry union to false. In this case, the Component must inform all of its children that they are invisible. The LayoutMgr is responsible for rendering a background colour in any regions of the Layer which are not filled by Components.

With rectangular Layers, by the time SetGeometry() is called on any of the Components, the LayerMgr has calculated the visible shape of the Layer, and the Layer is configured to clip drawing operations correctly. In this situation, the suspend argument to SetGeometry() is set to false, causing the Component to immediately re-paint itself with its new geometry.

If the Layer has a complex shape, it cannot know its new visible shape at the time SetGeometry() is called, so the suspend argument is set to true. This means that not only should the Component (and its children) not immediately re-paint, but also that it should not perform any other rendering operations until it is explicitly permitted to paint. If this condition was not met, the drawing operations would be clipped to the Layer's previous visible shape, and might therefore not appear, due to being outside the old shape, or may momentarily appear, only to be rapidly over-written, due to being inside the old shape, but outside the new one.

Once the shape of the Layer has been determined, the Components may be re-painted with their new geometries, by a call to their Paint() operations. Paint() is also used when the visible shape of the Layer changes due to changes in other Layers, or the relative positions of Layers. In this case, the Paint() operation is passed a shape representing the 'dirty' areas, so that unchanged regions are not unnecessarily re-painted.

The recursive propagation of SetGeometry() and Paint() calls raises an interesting question about threading. If a Component has more than one child Component, and if those Components are running on different machines from each other, then there will be a performance gain in calling the children's operations in parallel, from separate threads. If, on the other hand, the Components are co-located, the use of extra threads will possibly result in lower performance due to the thread switching overhead, especially on a single processor machine. Unfortunately, there is no standard method of detecting whether two objects are in the same address space as each other, let alone within different processes on the same machine. The CurrentLocation attribute of the LCObject interface, supported by all Components, does make it possible to detect co-location, but since retrieving the value of the attribute involves a network round-trip, it is not useful to retrieve it before every operation invocation. It is up to individual Component implementations to resolve this situation; section 7.1.4 evaluates the use of the two threading models by some simple Components.

4.7.2 Dynamic addition of Components

The standard Component interface provides no means for Components to be added to the Component tree once the tree has been initialised. Many seemingly dynamic situations can, in fact, be handled in a static manner. A 'tabbed' dialogue, for example, is one in which several different sets of Components may be selected, based upon a set of 'tabs'. When one tab is selected, the Components from the previous tab disappear, and a new set appear. Rather than dynamically removing the Components from the

```
interface EventHandler;

interface EventHandler {
  exception NoSuchKey {};
  exception WrongType {};

  void Event(in string key, in any val)
    raises (NoSuchKey, WrongType);

  struct Spec {
    EventHandler eh;
    string       key;
  };
};

interface StateProvider {
  exception NoSuchKey {};

  any GetState(in string key) raises (NoSuchKey);
};
```

Figure 4.9: Event and state transfer interfaces

old tab, and adding the Components from the new one, this is best implemented by statically installing all of the possible tabs as sub-Components of the master tabbed dialogue Component. The dialogue Component then always tells all but one of its sub-Components that they are invisible.

There are, of course, some applications, such as web browsers, which *do* need to be able to dynamically add Components. This may be achieved either by creating a Component which has extra operations for managing dynamic behaviour, or through the Event mechanism described below. When Components are added or removed, or if a Component changes its size range for some reason, the layout of Components may have to be updated. This is achieved through a call to the LayoutMgr's `ComponentChange()` operation which, in turn, causes a series of recursive calls to all of the involved Components' `ReSize()` operations.

4.7.3 Events and state

DAWS has two supplementary generic interfaces which provide standard methods of transferring state between Components. Both interfaces are shown in figure 4.9.

The `EventHandler` interface has two primary purposes. The first is to communicate shared state between Components which form one logical control, such as the linked scroll-bar and text box described above. In this case, both of the Components derive from the EventHandler interface. The second purpose is to communicate user interface events from Components back to the application. In this case, the application creates an EventHandler object (or an object derived from EventHandler) to be contacted by the Components making up the user interface. In both cases, EventHandler::Spec structures are passed as arguments to the Components.

The StateProvider interface provides what is, essentially, a generic means to access read-only attributes. It is primarily used by applications which sometimes need to access some aspect of the state of a Component, but are not interested in exactly when the state changes. One example is a check-box Component used to present a yes-or-no choice within a form. The application does not need to know how often the user toggles the state of the Component; instead, it queries the Component when the user submits the form by interacting with a different Component. Components which have 'attributes' which can be accessed through the StateProvider interface should also provide genuine IDL attributes, so that clients may access the Component's state with compile-time type safety, if they have access to the Component's IDL definitions.

4.8 Other window system facilities

This section provides an overview of some of the other facilities provided by DAWS. None of these facilities are integral to the design of DAWS, although they are required by a full-featured window system. The extensible nature of DAWS means that it is simple to add other, similar, services, or replace the ones described here.

4.8.1 Session management

Each desktop environment consisting of a set of overlapping Layers, managed by a LayerMgr, forms a DAWS *session*. Each user may maintain a number of different sessions. The user, or rather the DAWS system working on the user's behalf, must be able to find references to all of the top-level objects involved in a particular session, wherever the user may be. Once the object references of the primary objects have been found, the objects may be instructed to migrate to a location close to the user. In particular, migrating a session's Display object 'teleports' the session's imagery to a display visible to the user.

The LayerMgr interface provides operations which permit all of the other primary objects discussed so far to be found. However, it is useful, as demonstrated by the facilities outlined below, to be able to associate various other objects, such as a clipboard, with individual sessions. DAWS therefore defines a Session interface, which is derived from the ResourceFinder interface. The session's LayerMgr object may be found via an attribute of the Session interface. The ResourceFinder and ResourceFactory mechanisms permit Sessions to be extended to support any service required.

To simplify the user's interface to their different Sessions, they have a single SessionMgr object which allows them to find Session objects by name. The user thus only has to carry a single object reference, to their SessionMgr, with them, to permit them to access all of their Sessions wherever they may be.

The Session system purposefully avoids the enforced use of the naming service, so as to remain as flexible as possible. The implementation of the SessionMgr could make use of the naming service, but it does not have to. One significant reason for not using the naming service directly is that the user must be able to access information about their Sessions from anywhere in the world. In the absence of other security mechanisms, it is common to restrict access to a given instance of the naming service to the island it resides within. Both Session and SessionMgr objects may migrate to follow the user. It is particularly important to migrate Session objects, since they house some relatively dynamic objects.

4.8.2 Windows and window management

All of the discussion of overlapping regions has so far concentrated on Layers, and the Components contained within them. A useful *window* system obviously requires some user-level concept of windows, with title bars, some means of re-sizing them, and so on. Windows within DAWS are implemented through the Component mechanism, with coordination provided by a WindowMgr object. The WindowMgr is a Resource provided by the Session. To create a window, the application creates a tree of Components, just as described in section 4.7.1, with the top Component of type C_Window. The C_Window Component accepts a single Component as its contents, and may also possibly create a number of sub-Components as controls for the window. It is responsible for providing the imagery and interactive behaviour which supplies the user's experience of a window.

The WindowMgr is responsible for deciding where on screen to place new windows, either based upon some algorithm, or by receiving user input. It is also in charge of activating windows for input, based upon some policy such as focus-follows-mouse or click-to-focus. It may additionally intercept input so as to provide keyboard interfaces to input focus, and functions such as moving and iconifying windows. Window Components are able to support iconification by setting all of the Components contained within them to be invisible, and displaying icon imagery.

The details of how the window Component and WindowMgr perform their tasks are intentionally very loosely specified, so that DAWS can support a wide range of different policies and splits in functionality.

4.8.3 Text rendering

Flexible support for text rendering is clearly an important aspect of a window system. As with the rest of the system, the text rendering system used by DAWS attempts to be a flexible as possible. All text to be rendered in represented using the Unicode character encoding[Uni96], so characters from many different alphabets can potentially be rendered, and the specification of fonts is independent of any particular font format, and supports both bitmapped and scalable fonts. Fonts are specified by a set of *properties* indicating the style of the font (italic, bold, small-caps, fixed-width, etc.), whether it is scalable or bitmapped, whether is supports hinting (which makes scalable fonts look better at small sizes), whether an anti-aliased (smoothed) version is available, whether it supports kerning (reducing the space between pairs of letters, such as 'AV', which fit together), and what its design size is. Not all font rendering implementations necessarily support all of these possibilities.

Text rendering is performed by `TextRender` objects, which are top-level objects, as opposed to Resources of other objects. The Resource mechanism is not particularly suitable for finding objects which render text, since each font instance requires a different object; the Resource mechanism only permits a single object of a particular type to be bound to a particular name. A text rendering Resource would therefore have to have a number of sub-objects, which would further complicate object migration.

`TextRender` objects are created with `TextRenderFactory` objects. As well as operations to create and clone `TextRender` objects, the `TextRenderFactory` interface also has an operation to query which fonts are supported at the factory's location, and what their properties are. `TextRender` objects can be used in two different ways. The first, preferred, way is for the `TextRender` to render directly into a drawing Resource; the second is for it to return a map of pixels to the client. The second method is only really useful if the client wishes to perform some special effect on the rendered text.

Text can be rendered into a drawing Resource with either a solid or a transparent background. With a solid background, the client specifies both the foreground and background colours, and a rectangle containing the text is rendered; with a transparent background, only the text itself is rendered. For this to be possible, a DrawAlpha or DrawMask Resource must be available. Anti-aliased fonts, rendered via a DrawAlpha Resource, blend into whatever imagery was under the text.

Each `TextRender` object renders a single font with a single size into the drawing Resources of a single ResourceFinder; clients wishing to render different fonts or sizes must create multiple `TextRender` objects. `TextRender` objects are stateless in all other ways, so multiple clients may use a single object. This means, for example, that all of the Components within a Layer can use the same `TextRender` object to render all their text. Clients rendering into different ResourceFinders may not share a `TextRender`; if this was permitted, the `TextRender` would have to query the ResourceFinder on each rendering operation, which would be inefficient.

4.8.4 Clipboard

DAWS implements a clipboard in a very simple manner, with a Clipboard Resource supported by the Session object. The contents of the clipboard are represented by a single attribute, of type any. A read-only attribute allows clients to discover the typecode of the current clipboard item, so that resources are not wasted in transferring an item of a type which cannot be handled. This introduces a potential race condition, as the clipboard contents could change between a client checking the typecode, and it reading the item. This is not really a problem, since applications should only ever store data on the clipboard due to a request from the user, and the user cannot simultaneously perform cut and paste operations. Nevertheless, clients should cope gracefully with receiving a clipboard item of an unexpected type.

Simple, small, data items, like Unicode strings, are stored directly by the clipboard Resource, for ease of access. Storing larger items in this way would be wasteful of both network resources, and storage on the clipboard. Instead, the application placing the item creates an object which allows the data to be accessed, and places a reference to that object on the clipboard. Clients retrieving such a clipboard item must, of course, understand the interface used to access the data. Items placed directly on the clipboard survive beyond the life-time of the applications which placed them there; items represented by objects, however, potentially disappear when the application which created them exits. Applications may take some steps to avoid this if they wish, but clients should cope with the failure of an object reference retrieved from the clipboard.

Clients may register EventHandler objects with the clipboard, so that they are informed when the clipboard contents change. The content change message contains the typecode of the clipboard item. This permits Components which have placed an item on the clipboard to change their appearance when the item is no longer on the clipboard, and for Components to highlight themselves to indicate that they could accept the data currently on the clipboard. It also provides a simple means to create a clipboard viewer which always shows the current clipboard contents.

4.8.5 Pointer imagery

It is useful for applications to be able to change the appearance of the mouse pointer as it moves over different parts of the display, to indicate the actions which may be performed in each area. DAWS permits this through a Resource of the Display object, named PointerMgr. This Resource permits clients to associate different pointer images with shaped regions of the display, represented using the standard shape structure. It is then up to the display to change the pointer imagery as it moves from region to region.

To simplify Components' interface to the PointerMgr, the Layer objects may also support the PointerMgr Resource, providing coordinate translation, so that the Components do not need to know their Layer's current position, or update the display's

PointerMgr when the Layer moves.

The PointerMgr can be implemented either directly by the display, or may run in a separate location via the ResourceFactory mechanism. To be able to run a remote PointerMgr, the display must support the `PointerSet` Resource, which permits a client to directly set the pointer imagery. The remote PointerMgr can then register itself with the display's Mouse::Server Resource, and update the pointer image via the PointerSet Resource.

4.8.6 Configuration database

Another Resource which is provided by the `Session` object is a configuration database, similar to the resources database used by X. It basically provides a mapping from hierarchical names to values. Both names and values are simple strings, meaning that the database can be stored, and edited, as a simple text file. Elements within the naming hierarchy are separated by '`/`' characters. The top-level element of the name is the name of the application. Following the application name can be any number of elements representing logical sections within the application. After that, come three more elements: the name of a Component within the section, the IDL type of the Component, and finally the property of the Component which is being stored. So, for example, within a calendar application, the font used on the button labelled 'Add' on the form used to enter appointments may be named '`Cal/AptForm/AddBtn/C_Button/FontName`'.

Components query the database for each of their properties, using the full name. It would be extremely inefficient and inconvenient to have to record a value for every single property within the database, so names within the database may contain wild-card characters. An asterisk, '`*`', will be match any number of elements within a name query, except the last element. So the above string would match '`*/FontName`', '`*/C_Button/FontName`', and '`Cal/*/FontName`', but not '`*/C_Button/*`' (which is an invalid entry).

If more than one database entry matches the requested name, then the 'best' match must be returned. Higher-level (i.e. left-most) elements of the name are considered most important, so the entry containing the highest-level element is used. If two or more matching entries contain the same highest level entry, then they are chosen between based upon which one has the next highest entry, and so on. So, in the above example, the '`Cal/*/FontName`' entry would be returned. If the database also contained '`Cal/*/AddBtn/*/FontName`' and '`Cal/AptForm/*/FontName`' then the last of these entries would be returned. This means that the configuration database can contain global defaults for various properties, and properties within named Components, with exceptions for particular applications.

4.9 Summary

This chapter has detailed the design of DAWS as it fits within the framework described in Chapter 3. The design is centred around three basic principles:

4.9.1 Low latency

As explained in Chapter 2, window systems are highly dependent on the effects of latency. The following design decisions serve to limit the latency of the most important object communications:

- The important aspect of latency is the time taken for an event from propagate from the user to the object which handles the event, and for the results of the event to propagate back to the user's display. By minimising latency at all stages, and limiting the number of stages, the total latency is minimised.
- Low-level input events are transmitted using specific-purpose interfaces, rather than through a generic event mechanism, so as to reduce marshalling delays. This approach also simplifies the event handling code.
- The low-level events are passed directly from the event producer to the event consumer, avoiding any intermediate objects. This places a greater burden on the event producer—the mouse server must perform position translations for different clients, for example—but greatly reduces latency.
- A shaped region representation which is more efficient than that used by many window systems saves processing time, and thus offsets some of the effect of additional network latency, especially for complex shapes.
- GUI Components are able to communicate directly with each other using the `EventHandler` and `StateProvider` interfaces, avoiding network round-trips to the main application code.

4.9.2 Platform flexibility

DAWS has been designed to be portable to a very wide range of devices, from very 'thin' network-attached displays and input devices, though to high-powered general-purpose workstations:

- The Resource mechanism permits top-level objects such as displays to support a range of different facilities. This means that high-powered devices can support complex Resources, which low-powered devices only support simple Resources.

- The Resource mechanism also permits special-purpose Resources to be defined, so that clients can make use of esoteric hardware.
- The ResourceFactory mechanism allows objects to have remote Resources, to 'fill in' for Resources not available locally.
- The lowest-level drawing operations use a resolution-dependent integer-based imaging model, so as not to place too great a burden on simple displays. Higher-level drawing operations may be supported though the Resource mechanism.

4.9.3 Adaptability and extensibility

DAWS has also been designed to adapt to different types of device, and to extend to new types of device and new interfaces to existing devices:

- All major DAWS objects are able to migrate, so they can adapt to changing circumstances.
- The ResourceFinder and ResourceFactory mechanisms provide a means of dynamically extending an object's interface.
- The system of user interface Components permits a wide range of different GUI elements, with differing levels of complexity, to be created.
- Services such as window management, text rendering, and clipboard support, which are integral parts of other window systems, are implemented using separate objects and Resources within DAWS, so it is easy to extend or replace them.

The next two chapters describe some implementation details of parts of DAWS. Chapter 5 explains how omniORB has been extended to support migration of CORBA objects. Chapter 6 shows how some of the objects described in this chapter have been implemented, and explores some of the challenges of writing highly distributed programs.

5

Object Migration

DAWS relies heavily on the assumption that objects are able to migrate from location to location. Section 3.5.3 explained that DAWS objects are responsible for transferring their own state when they migrate, so the issue to be solved is how clients of an object track the object as it moves. This chapter discusses different methods which could be used to achieve this, and details the design and implementation of one particular scheme. The implementation involves additions to omniORB, the CORBA implementation used by DAWS.

Supporting migration of CORBA objects is challenging because object references contain the host name and port number of the server at which the object resides. Once an object reference has been made available, it must remain valid for the entire life time of the object to which it refers.

The migration scheme was designed for DAWS, so it is most suited to the type of system which involves large numbers of highly distributed transient objects, each moving in a relatively independent manner. Systems made up of large numbers of possibly persistent objects, in a small number of locations, which may wish to migrate *en masse* would benefit from a different approach. It should also be noted that DAWS does not rely on this particular migration scheme in any way—all it requires is that it can issue object references which remain valid when objects migrate—so it would be simple to replace this scheme with another.

> Remember that this chapter only covers the functionality required to move the client's view of an object from one location to another—that is, the client stops using an object implementation at one location and starts using an implementation at another location. The means by which the implementation and state of the object are transferred is a separate issue.

5.1 Background

Several commercial ORBs, notably HP's ORBPlus, claim to support the CORBA Life Cycle service, which implies support for object migration. What these products actually provide are basic implementations of the FactoryFinder and GenericFactory interfaces; they provide no support for object migration. Some ORBs do permit objects, or groups of objects, to move within a limited domain with support from their implementation repositories. This does not provide a scalable solution and also makes it impossible to migrate object implementations between different ORBs, as implementation repositories are ORB-specific. The basic approach is that the IOR of an object which is to migrate is set to contain the address and port number of the implementation repository, rather than a direct reference to the object. The implementation repository is then responsible for forwarding clients to the actual object.

Henning[Hen98] describes, in more detail, a number of approaches to object migration which may be provided by an ORB's implementation repository. He claims that, due to the way implementation repositories work, it is only really feasible to migrate all objects registered with a particular object adaptor—migration of individual objects places an unreasonable burden on the repository's database—and migration between location domains (i.e. islands) is difficult. He maintains, incorrectly, that if an object migrates between location domains a number of times, there must be a build-up of state at every location the object visits. In fact, the migration mechanism described in this chapter could be extended to work with implementation repositories, and would not involve a state build-up.

Hauck, van Steen and Tanenbaum[HvST97] describe an interesting hierarchical architecture for locating objects in a global system. It works by segmenting the world into nested regions to form its hierarchy, and splitting the key space to reduce the burden on root servers. It assumes firstly that objects most commonly migrate to nearby regions, thus reducing the update rate at the root servers, and secondly that object creation and deletion are rare, as all creations and deletions have to be registered with the root. The first is true for DAWS whereas the second is not. It is possible to integrate this scheme with CORBA by running a special service on each machine involved in the system, and creating IORs which refer to the local loop-back interface[WH99]. Alternatively, it could be coupled with some of Henning's implementation repository ideas.

In a system where objects can move from location to location, it may seem sensible to make use of portable languages such as Java[GM95]. Indeed there are many non-CORBA-based systems which make productive use of mobile code, either as their only relocation mechanism[BC96, BHR$^+$98], or in combination with other services such as traders[BHB96]. It is important to realise that mobile code does not, in fact, solve the problem at hand—it helps with system flexibility since servers no longer require compiled-in support for all objects they can serve, but it does not aid with the transparent redirection of clients' requests. The Voyager system[Obj97] permits remote Java objects to migrate in such a way that their clients transparently follow them, using a combination of forwarding agents and a directory service. A number of other agent-based systems implement varying-levels of object tracking, in conjunction with code and state transfer[Bat98].

5.2 Implementing fine-grain object tracking

The existing approaches to object migration cannot cope with objects which regularly migrate on an individual basis, and are regularly created and deleted[Hen98]. This section evaluates a number of different possible schemes which support object tracking in support of migration. Note that the first three approaches are independent of CORBA and would work with any remote object system.

1. Indirect all object operations through a proxy object which forwards requests to the actual object.

   This would be very simple to implement, requiring no additional ORB facilities. Mobile IP[Per96] uses a system like this to support mobile hardware devices. Unfortunately, since one of the main reasons for migrating objects is to reduce network latency, this scheme is inappropriate for DAWS.

2. Register all objects, and their locations, with some sort of naming service.

   With this approach, whenever an object moved its clients would detect that it no longer existed at its old location, and find its new location from the naming service. This would require extensive support within clients, and would require that all objects were given names. For such a system to scale, a distributed location service like Hauck et al.'s would be necessary.

3. Make object references bi-directional: each object keeps a list of clients which hold references to it.

   When an object migrated, it would inform every client which held a reference to it. This approach would mean that all clients would have to be CORBA objects themselves, and would make all object implementations much more complex. It would also be highly susceptible to failures due to the inter-dependence of all objects.

4. Make use of the object location support in GIOP (and so IIOP) whereby any server agent may reply to an operation invocation with a LOCATION_FORWARD message containing the object reference of the actual object.

   One way to make use of this mechanism would be to ensure that every time an object migrated, it 'left behind' an agent which forwarded clients to the object's new location. This would result in a trail of forwarding agents at every location the object had ever occupied, leading to intolerance to faults, and difficulties with garbage collection. The TAO ORB[Sch98] supports migration through this mechanism[KP98, Kir98][1].

5. For each object, nominate a *home* location which is guaranteed to know the current location of the object.

   In this scheme, the home location either provides the object itself, or contains an agent which sends LOCATION_FORWARD messages referencing the actual object. Clients remember the home location, and consult it whenever the object ceases to exist in the location they were using. Whenever an object migrates, it informs the home location of its new location.

The design presented in this chapter uses the final approach. One disadvantage of this approach (and of most of the other approaches) is that it depends on the object's home location being available at all times, thus having a single point of failure. This is not considered too serious a problem, since each object may have a different home location—the failure of a single machine only affects objects with their home locations on that machine, not all objects in the system. In a window system, where the state of most objects can be re-constructed from state held by other objects, total fault tolerance is not necessary. In a system where this *was* a major concern, the home location machine could be replaced by a cluster of machines. The system *is* tolerant of transient problems at the home location, as long as it restores its state after crashes.

Another difficulty arises if the network becomes *partitioned*, so that the home location cannot be reached. This situation prevents an object from changing location, even if it can communicate with its desired destination. It also means that new clients cannot bind to the object, even if they can communicate directly with it, since they are unable to find its current reference from the home location. Existing clients of the object are not affected, however. The fact that CORBA object references contain a host name makes this situation insoluble without making clients much more aware of object migrations. Given that DAWS applications generally leave at least parts of their functionality in the user's home island, there would be wider difficulties if the network became partitioned. DAWS assumes that any network faults will be transient.

The system could be even more flexible if it was combined with the implementation repository approaches described by Henning, permitting efficient migration of both single objects and groups.

[1]TAO's migration support was added around one year after the system described in this chapter was released as part of omniORB.

5.2.1 Object adaptors

The migration implementation described in this chapter was made when the Basic Object Adaptor (BOA) was the standard CORBA object adaptor[OMG96]. The BOA has no interface to allow implementation code to send LOCATION_FORWARD messages, and the exact arrangement of C++ objects used during a call from a client to an object is not fully specified. This means that different ORBs have different mechanisms for sending LOCATION_FORWARD messages (or no mechanism at all), and the way in which such messages are handled also varies.

The newer Portable Object Adaptor (POA) specification[OMG98a], which is not yet supported by many ORBs, addresses many of these issues, although there are still some under-specified areas. Section 5.8 discusses the ways in which the scheme described in this chapter could be converted to work with the POA.

5.3 Requirements and design

This section details the services required by an object migration scheme based upon home locations, and presents a high-level design which addresses the requirements. The requirements and design also place a number of requirements on the ORB.

5.3.1 Requirements for supporting objects which may migrate

As far as objects implementations are concerned, the following services are required:

- Creating an object (at its home location).
- Migrating an object from its home location to a remote location, leaving behind a 'forwarding agent' which emits LOCATION_FORWARD messages pointing to the new location.
- Migrating an object from one non-home location to another, and informing the home location of the move.
- Migrating an object from a non-home location back to its home location.
- Removing an object at its home location.
- Removing an object at a non-home location, informing the home location.

Note that, apart from creating the object, the object implementation itself must be able to trigger these activities. Also, remember that migrating an object involves informing the system that it should use a different object implementation in future; creating the new object and transferring object state are separate issues.

As far as clients are concerned, the system must handle LOCATION_FORWARD messages, including cases where the object moves into or out of the client's address space.

The system should cache the object's current location, rather than always consulting the home location, otherwise there would be very little benefit to migrating an object. If a client which has been forwarded receives a communication failure or OBJECT_NOT_EXIST exception when attempting to communicate with the object, it should revert to contacting the home location.

The remaining important requirement is that whenever an object reference to a migrated object is exported, either by the object implementation itself or by a client holding a reference, the exported reference is to the object's home location, rather than to its current location.

5.3.2 Design

The high-level design is very simple. All of the functionality required by clients must be provided by the ORB; the mechanism by which an object at the home location is replaced by a forwarding agent is also ORB-specific. The only area of the design which is not system-specific is the means by which the forwarding agent is kept up-to-date with the location(s) of the object(s) for which it is responsible.

Each object which has migrated has an associated LifeCycleInfo[2] object which always remains at the object's home location. Its interface is:

```
// idl
interface LifeCycleInfo {
  void reportMove(in Object obj);
  void reportRemove();
  readonly attribute Object homeObject;
};
```

When an object implementation wishes to migrate, it calls its own LifeCycleInfo's reportMove() operation, giving the object reference of the destination object. The LifeCycleInfo implementation is responsible for updating the object's forwarding agent. The LifeCycleInfo object need not be created until the object first migrates away from its home location.

When an object migrates to a new location, it passes a reference to its LifeCycleInfo object to the factory at the new location. The factory is responsible for informing the ORB that when the object implementation exports its own object reference (with a call to its _this() member function), the reference of the home location is actually exported. The homeObject attribute returns a reference to the home location.

When an object is to be removed, the home location is informed by a call to the reportRemove() operation. This triggers the deletion of the forwarding agent and the LifeCycleInfo object itself.

[2] In the omniORB implementation, this interface is actually named omniLifeCycleInfo.

Concurrency control

DAWS operates within a multi-threaded environment, so it is possible for multiple operations to be in progress at once. Objects can also have internal threads. Care must therefore be taken not to move or remove an object while other threads are running within it.

The ORB and migration implementation are responsible for protecting their own data structures, but object implementations must also be protected. Such concurrency control is left to the implementation, since only it can know which of its operations may perform operations which affect its life cycle. Also, it may be possible to create specialised concurrency control which is more efficient than a general solution could be.

A default solution with a simple programmers' interface has been created, so implementors do not have to worry about concurrency control if they do not wish to. It is described in section 5.6.2.

5.4 omniORB operation

This section outlines the internal structure of omniORB. It highlights the areas which must be extended or modified to support the chosen migration scheme.

5.4.1 Generated classes

As was explained in section 3.4.2, static invocation requires that an IDL compiler generates stub and skeleton code for each entity defined in IDL. With no migration support, given an IDL interface `T`, omniORB's IDL compiler generates the C++ classes shown in figure 5.1. These classes follow the requirements of the BOA specification.

The `omniObject` and `CORBA::Object` classes define operations and state common to all CORBA objects. They are implemented within the omniORB library. The generated `T` is an abstract class defining a member function for each operation declared in the IDL definition of `T`, as well as static functions to narrow and duplicate object references. `_nil_T` is used for nil object references—it just throws `BAD_OPERATION` exceptions for each operation of `T`.

`_sk_T` is the implementation skeleton abstract class. It defines a `dispatch()` member function which calls the correct member function for each of the operations declared in `T`. This function is called by the ORB when it receives a request for one of `T`'s operations. It is up to the user-supplied servant (implementation) class, `T_i`[3], to actually implement the member functions. When a client receives an object reference to an

[3]The user may name this class anything they wish; it is only named `T_i` by convention.

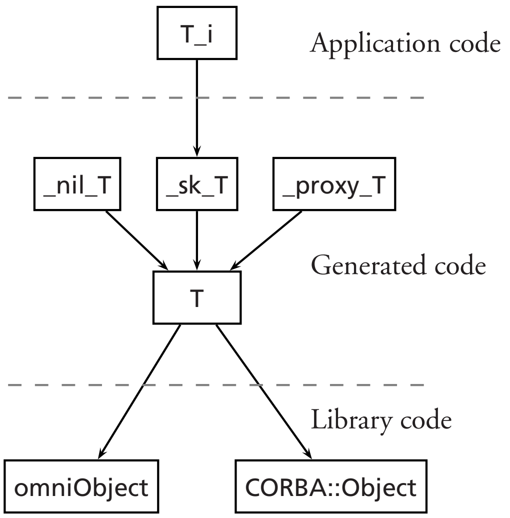


Figure 5.1: Hierarchy of classes used for an IDL interface T

object in its own address space, it gets a pointer to the actual T_i object as a local-case optimisation.

When a client acquires an object reference to an object in a remote address space, it actually receives a pointer to an object of type _proxy_T. For each operation of T, _proxy_T has a member function which performs the necessary steps, via the ORB, to invoke the operation at the remote location, and return the result.

Figure 5.2 illustrates the chain of objects used when a client holds a reference to a CORBA object.

5.4.2 Interface inheritance

If the IDL interface T derives from one or more other interfaces, each of the T classes shown in figure 5.1 derives from the corresponding classes from the other interfaces. So, for example, if interface T derives from interface U, class T derives from class U, _proxy_T derives from _proxy_U, and so on.

5.4.3 Support for LOCATION_FORWARD messages

When a _proxy_T object tries to perform a remote invocation, it receives one of the following IIOP messages:

- NO_EXCEPTION—the operation completed successfully.
- USER_EXCEPTION—a user-defined exception was raised.
- SYSTEM_EXCEPTION—a system exception was raised.

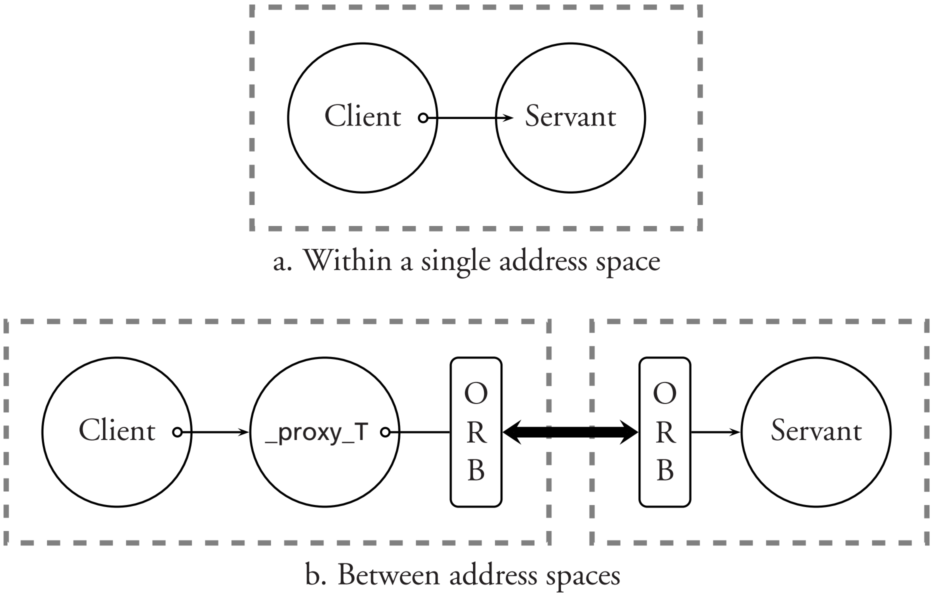


Figure 5.2: Chain of C++ objects used to represent a CORBA object

- LOCATION_FORWARD—a different object reference should be used in future.

In the case of a LOCATION_FORWARD, the _proxy_T object re-configures itself to send invocations to the new object, and re-tries the invocation.

If an object has been previously forwarded, and an invocation results in an OBJECT_NOT_EXIST or COMM_FAILURE system exception, the _proxy_T resets itself to use its original object reference, and throws a TRANSIENT exception to the caller. It does this, rather than re-trying the invocation, because to do so may result in repeated forwarding to the same failed location. The client may re-try the invocation a number of times if it wishes[4].

If a proxy object receives a LOCATION_FORWARD to an object in its own address space, there is no way for future invocations to be made directly to the servant object—invocations still go via TCP/IP, through the loop-back interface. Similarly, since clients in the same address space as an object do not use _proxy_T objects, but hold pointers directly to the object, they cannot be forwarded if the object moves away.

5.4.4 Forwarding agents

omniORB provides no interface to enable the creation of forwarding agents. Even if there was an interface, the fact that clients in the same address space as the object hold

[4]omniORB provides a default TRANSIENT exception handler which re-tries invocations with exponential back-off.

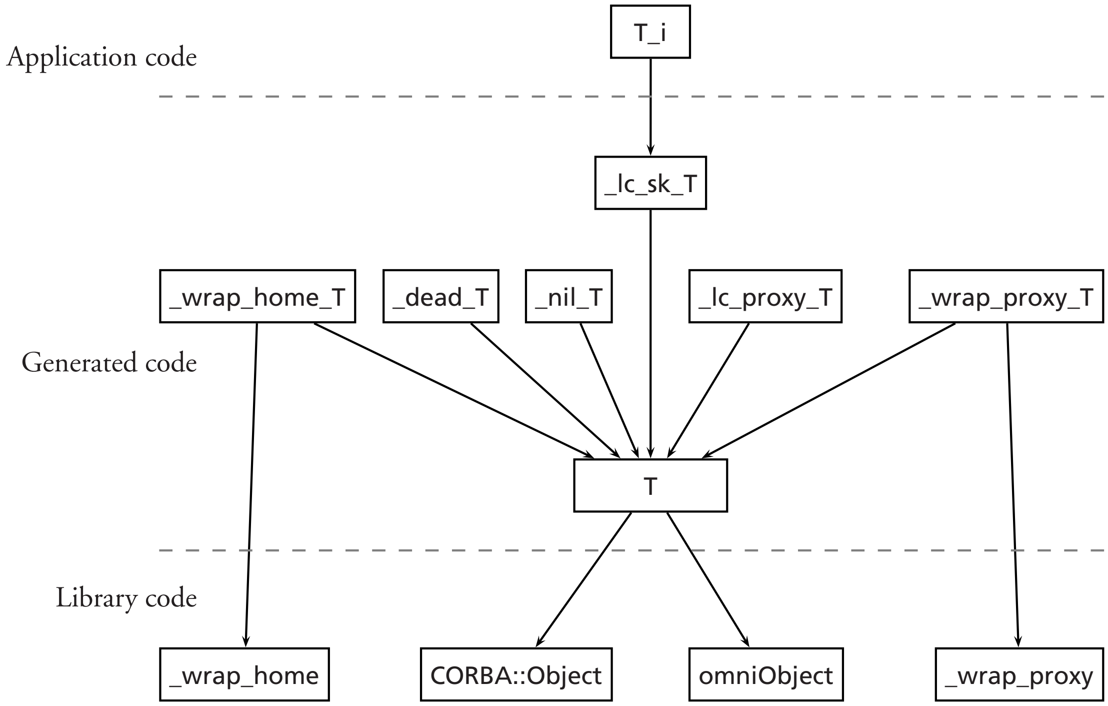


Figure 5.3: Hierarchy of classes used to support life cycle operations on interface `T`.

pointers directly to the servant means that it would not be possible to replace the servant with a forwarding agent. Similarly, the ORB itself does not permit one servant to be replaced by another.

5.5 Implementing migration support

omniORB's IDL compiler has been extended to generate the extra proprietary classes shown in figure 5.3. `_lc_proxy_T` and `_lc_sk_T` replace `_proxy_T` and `_sk_T`. The following sections explain the purposes of each class. The programmers' interface to the system is described in section 5.6.

The implementation works by providing C++ 'wrapper' objects around equivalents of the normal omniORB objects. This extra level of indirection permits servants to be replaced with forwarding agents, and proxies to be replaced with servants, as required.

5.5.1 The home location

The home location is the location at which an object is first created. For the entire existence of the CORBA object with interface `T` there is an object, of class `_wrap_home_T`, at the home location. This object keeps track of the location of the CORBA object, and correctly forwards invocations to it.

The _wrap_home_T class

All client operation invocations on a CORBA object of type T resolve to calls to the member functions of the _wrap_home_T object. Remote invocations result in calls to the dispatch() function; local clients hold a direct pointer to the _wrap_home_T object, so _wrap_home_T has a member function for each operation declared in interface T.

The _wrap_home_T object works by keeping two object references: _actual_T, of type T_ptr, is a reference to the implementation object, which may be local or remote; _dispatcher, of class omniObject, is a local object which provides a dispatch() method.

_wrap_home_T's own dispatch() function simply passes calls to the dispatcher object with the following code:

```
// C++ in class _wrap_home_T
CORBA::Boolean dispatch(GIOP_S &s,
                        const char *op,
                        CORBA::Boolean r) {
  return _dispatcher->dispatch(s, op, r);
}
```

If T has an operation defined, for example:

```
// idl
interface T {
  ...
  long op(in long arg);
  ...
};
```

then _wrap_home_T defines a function, which is only called by clients in the same address space as the _wrap_home_T, to pass the call on to the actual object:

```
// C++ in class _wrap_home_T
CORBA::Long op(CORBA::Long arg) {
  return _actual_T->op(arg);
}
```

When the implementation object is still at its home location, the situation is as shown in figure 5.4. Both _actual_T and _dispatcher (indicated in the figure by ‘a’ and ‘d’ respectively) are pointers directly to the implementation object (which is derived from _lc_sk_T. This means that both local and remote client invocations go to the implementation object with a single extra C++ method invocation.

When the implementation object has migrated to another location, _actual_T is a pointer to a proxy to the object, so local client invocations now go via the network.

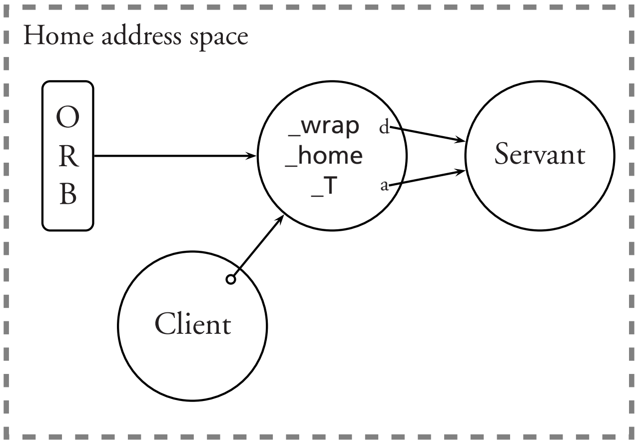


Figure 5.4: Home location before object migration

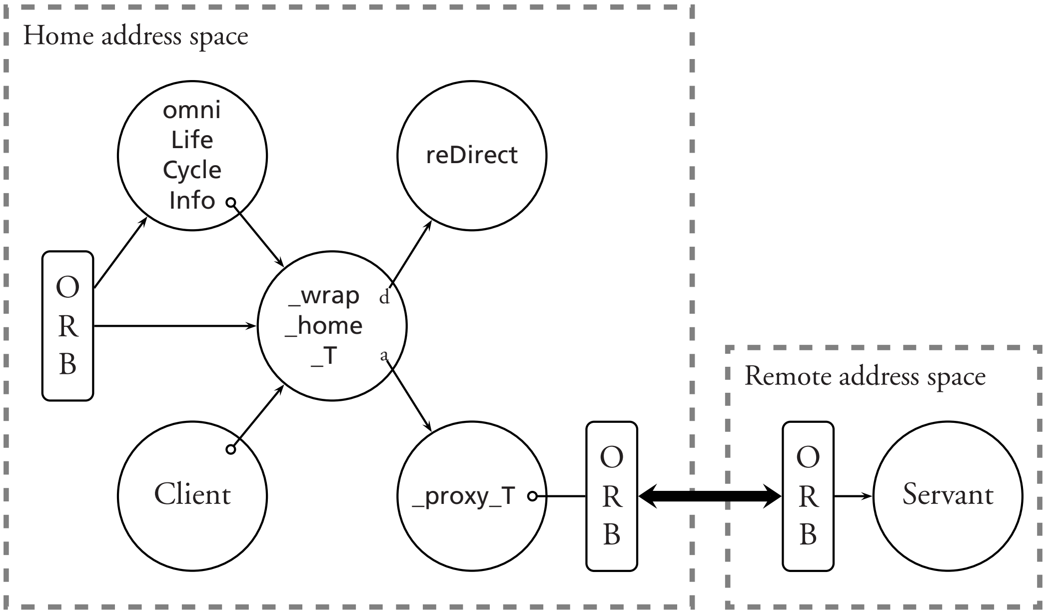


Figure 5.5: Home location after object migration

_dispatcher is a pointer to a forwarding agent of class reDirect, an omniORB-specific object which replies to all calls to its dispatch() function with a LOCATION_FORWARD message giving the object's current location. This configuration is shown in figure 5.5.

It is interesting to note that invocations from remote clients to moved objects are therefore slightly more efficient that ones to objects still at their home location (after the LOCATION_FORWARD message has been handled once), because the extra function call through the _wrap_home_T object is avoided. This situation could be avoided by always using the location forwarding mechanism, even for home location objects; this is not done because handling a LOCATION_FORWARD message is quite a time-consuming process, and because the overhead of the extra function call is very small compared to the total time spent processing a remote method call.

Interface inheritance

If the interface T derives from an interface U, then the class _wrap_home_T derives from the class _wrap_home_U. Both classes may use the same _dispatcher variable, so it is declared in the base class _wrap_home. Ideally, the _actual reference could be shared in the same way. Unfortunately, before the introduction of C++ run-time type information (RTTI), C++'s type system was unable to cope with this. Not all compilers support RTTI yet, so each class must define its own _actual_* variable. The _actual_* references of every class in the hierarchy must be updated each time the implementation object migrates. Even when RTTI is available on all compilers, it may be sensible to keep this arrangement since RTTI can impose a significant performance overhead.

Controlling the _wrap_home_T object

As explained in section 5.3.2, the home location's use forwarding agent is controlled by an object with interface omniLifeCycleInfo.

When an object is first migrated, its _wrap_home_T object creates an omniLifeCycleInfo object which remains in existence for the rest of the CORBA object's life. The omniLifeCycleInfo's reportMove() and reportRemove() operations call functions named _move() and _remove() on the _wrap_home object. The homeObject attribute is the object reference of the _wrap_home object which the omniLifeCycleInfo object controls.

_move() and _remove() are abstract member functions of the _wrap_home base class; _wrap_home_T defines _move() to update the _actual references and create a reDirect object for the new object location. _remove() clears up everything to do with the object, as described in Section 5.5.3.

5.5.2 Proxies to remote objects

Clients in a different address space to an object's home location use a proxy object to perform operation invocations over the network. As long as the implementation never migrates into the same address space as a client, the existing _proxy_T class, shown in figure 5.2, is sufficient to follow the object as it moves from location to location. However, if it is possible that the implementation *will* migrate into the same address space as the client, a different mechanism, similar to that for home location clients, is necessary.

When a client receives an object reference to a remote object with interface T, an object of class _wrap_proxy_T is created. This, like _wrap_home_T, has a member vari-

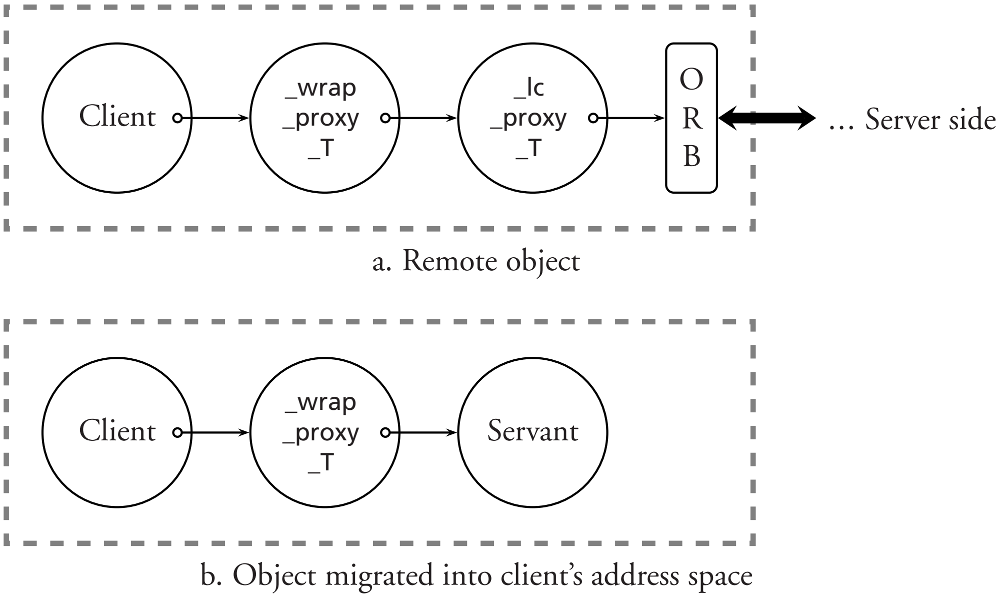


Figure 5.6: Configurations of clients at non-home locations

able _actual_T. As with the _wrap_home_T, each class within the inheritance hierarchy maintains its own _actual_∗ reference. The two configurations of the _wrap_proxy_T are shown in figure 5.6.

Initially, the _actual references point to an object of class _lc_proxy_T, which normally acts in the same way as a _proxy_T. Upon receiving a LOCATION_FORWARD message, an _lc_proxy_T behaves differently—if the object forwarded to is remote, a new _lc_proxy_T object is created, the _actual references in the _wrap_proxy_T are updated, and the invocation is re-tried. If the object is local, the _actual references are set to point to the servant object itself.

Migrating an object out of a non-home address space

At the home location, all clients which hold a reference to the object hold a pointer to the same _wrap_home_T object. At other locations, this is not necessarily the case—due to the way the ORB handles references to remote objects, different clients holding references to the same object may be using different _wrap_proxy_T objects. After an object has been forwarded into the address space of a number of clients, any clients which have invoked operations on it have _wrap_proxy_T objects which point directly to the implementation object. When the object migrates back out of the address space, these pointers become invalid, and must be reset.

Whenever a _wrap_proxy_T is set to point to a local object, it adds itself to a hash table based on the object key of the implementation object. The implementation can then reset all _wrap_proxy_T objects with pointers to it when it migrates to another address space. This functionality is common to all proxy wrappers, so it is implemented in the _wrap_proxy base class.

5.5.3 Object removal

Removing an object at its home location is not always as simple as just deleting the servant object, its wrapper, and the omniLifeCycleInfo object: once an object has been removed, all clients attempting invocations should receive OBJECT_NOT_EXIST exceptions. However, clients in the home location address space hold direct pointers to the object's _wrap_home_T object. If the _wrap_home_T was deleted, then the next time such clients attempted to access the object, the result would be undefined—the program would probably crash.

Therefore, when an object is removed, all objects at the home location except the _wrap_home_T, are deleted (using BOA::dispose()). A new object of class _dead_T is created, and references to it are placed in the _actual and _dispatcher values of the _wrap_home_T. The _dead_T object simply throws an OBJECT_NOT_EXIST exception for all of its member functions. BOA::dispose() is then called on the _wrap_home_T object, so that when its reference count becomes zero (i.e. all local clients have released their object references), it is deleted. _wrap_home_T's destructor disposes of the _dead_T object, finally clearing all storage related to the CORBA object.

Removing an object which has migrated is essentially the same—the remote servant is deleted, then the home location objects are deleted as before.

5.5.4 Exporting object references

It is important that whenever a servant or a client tries to export a reference to an object which may move, it actually exports a reference to the object's _wrap_home_T object. For implementations, _lc_sk_T defines the standard _this() function to return a reference to the _wrap_home_T object. Migrated objects acquire this reference from the omniLifeCycleInfo's homeObject attribute.

Clients in an object's home location already hold a pointer to the object's _wrap_home_T object, so no special activity is required. Clients with a reference to a remote object hold a pointer to a _wrap_proxy_T object. When the _wrap_proxy_T is created, it registers itself with the ORB as a proxy to the _wrap_home_T object, so if the client exports the reference they hold, the object reference of the _wrap_home_T is sent.

5.6 Programmer's interface

There are three main parts to the programmer's interface—creation of mobile objects by factories, concurrency control within operations, and control of migration and dele-

```
// EchoLag.idl

interface omniLifeCycleInfo;

interface EchoLagFactory;

interface EchoLag {
  string echoStringLag(in string str);
  void move(in EchoLagFactory there);
  void remove();
};

interface EchoLagFactory {
  EchoLag new_EchoLag();
  EchoLag clone_EchoLag(in string str,
                        in omniLifeCycleInfo li);
};
```

Figure 5.7: Example IDL definitions for an EchoLag interface which can move from location to location.

tion activities.

The following sections make use of an example interface for an EchoLag, shown in figure 5.7. It declares the operation echoStringLag() which takes a string as its argument, and returns the string it received on its previous call. So, on its first call it returns an undefined string, on its second call it returns the string it was given on the first call, and so on. This rather contrived example demonstrates the migration of an object which has some state.

5.6.1 Creating mobile objects

As figure 5.7 shows, the EchoLagFactory interface declares two different operations for creating EchoLags—one for new EchoLags, and one for cloning EchoLags during migration. Creating a new EchoLag is just the same as it would be without the migration extensions:

```
EchoLag_ptr
EchoLagFactory_i::new_EchoLag() {
  EchoLag_i *elag = new EchoLag_i("First call");
  elag->_obj_is_ready(elag->_boa());
  return elag->_this();
}
```

As explained in section 5.5.4, the _this() member function is defined in the EchoLag skeleton class, _lc_sk_EchoLag. It creates the necessary _wrap_home_EchoLag and omniLifeCycleInfo objects, and returns a reference to the new _wrap_home_EchoLag object.

Cloning an EchoLag during migration is a little more involved:

```
EchoLag_ptr
EchoLagFactory_i::clone_EchoLag(const char *str,
                                omniLifeCycleInfo_ptr li) {
  EchoLag_i *elag = new EchoLag_i(str);
  elag->_set_lifecycle(li);
  elag->_obj_is_ready(elag->_boa());
  return EchoLag::_duplicate(elag);
}
```

The important steps are to call _set_lifecycle(), defined in _lc_sk_EchoLag, and not to call _this(). _set_lifecycle() identifies the new EchoLag as a moved object: it calls homeObject() on the omniLifeCycleInfo object and uses the result as the return value of future calls to _this(). Using the result of _this() from the operation would therefore pass the object reference of the home location wrapper, rather than the new servant object.

5.6.2 Concurrency control

To aid with concurrency control within implementations, a class _threadControl is defined. It has four member functions: _beginOp() and _endOp() should be called at the beginning and end of a normal operation respectively; _beginLC() and _endLC() should be called at the beginning and end of an operation which affects the object's life cycle, such as moving or removing it.

If _beginLC() is called when one or more threads are within a _beginOp() / _endOp() region, it blocks until all threads have executed the _endOp(). If a thread has called _beginLC() but not yet called _endLC(), calls by other threads to either _beginOp() or _beginLC() throw TRANSIENT exceptions.

So that the implementation does not have to guard all possible exit points from a function, two helper classes, ThreadOp and ThreadLC are defined. Their constructors call _beginOp() or _beginLC() and their destructors call _endOp() or _endLC(), so they may be used as follows:

```
// The class eg is derived from _threadControl
CORBA::Long eg::operation() {
  ThreadOp _t(this);   // Don't care about the name.
  if (...some condition...)
    throw ERROR;
  else
    return 42;
}
```

5.6.3 Migration and deletion operations

To implement a migration operation, a reference to the object's omniLifeCycleInfo object must be passed to the object factory; _lc_sk_T has a member function _get_lifecycle() for this purpose.

_lc_sk_T defines _move() and _remove() functions which use the omniLifeCycleInfo object to report the move or remove operation. In the EchoLag example, the object's move() operation is implemented as:

```
void
EchoLag_i::move(EchoLagFactory_ptr there) {
  ThreadLC _t(this);

  // Create a clone of ourselves:
  char *p = CORBA::string_dup(s);
  EchoLag_var newEchoLag = there->clone_EchoLag(p, _get_lifecycle());

  // Tell the system to do the migration:
  _move(newEchoLag);

  // Dispose of the old servant:
  CORBA::BOA::getBOA()->dispose(this);
}
```

5.6.4 Proxy objects

By default, clients assume that implementation objects will never move into their own address space, and so use the normal _proxy_T objects, thus avoiding the overhead associated with _wrap_proxy_Ts. The client must inform the ORB which types of objects may migrate into their address space.

5.7 Performance measurements

A good measurement of the delay involved in an ORB is the time taken to echo a null string, since that indicates the operation invocation overhead, rather than the overhead of marshalling long arguments. Table 5.1 shows the average time taken in various configurations, with and without the migration extensions. Since a string echo object has no internal state, it is not necessary to perform any concurrency control within the echo operation. The 'LC' column indicates whether the migration extensions were used or not; the 'TC' column indicates whether thread control was used. To indicate how much overhead the standard thread control mechanism involves, times are given for implementation both with and without it. Times given are in microseconds per invocation, averaged over 10000 invocations.

Situation	LC	TC	Time (μs)
Client and server co-located	No		3.4
	Yes	No	3.5
	Yes	Yes	5.6
Server moved into client's address space	Yes	No	3.6
	Yes	Yes	5.7
Cross process via TCP/IP	—	—	440
TCP/IP over Ethernet	—	—	690

Table 5.1: Average time taken to echo a null string with and without migration extensions

The table shows that the majority of the overhead involved in using the migration extensions is in fact in the thread control—the overhead of the extensions alone is at the limit of what can be measured. The delay (and variance in delay) incurred by performing operations between address spaces is so large that the extra overhead imposed by the extensions is undetectable. Even the overhead due to the first invocation handling a LOCATION_FORWARD message is lost—it is only about the time of a single invocation. Clearly this overhead would be significant if it was normal for only a very small number of operations to be invoked on an object before moving on to another one. This is fortunately not the case in most parts of DAWS.

5.8 The Portable Object Adaptor

As mentioned in section 5.2.1, the POA specification addresses many of the object migration issues discussed in this chapter. It requires that there is more separation between the objects which a client accesses and the objects within the server, especially when client and server share an address space. It is not permitted, under the POA, for a client to hold a direct pointer to a servant. This means that all ORBs using the POA should cope properly when objects migrate into a client's address space.

The POA also provides a standard interface to sending LOCATION_FORWARD messages, although it does not fit especially well with the present design. If the POA receives a request for an object which is not currently registered, it can contact a ServantManager object which is responsible for finding the object. One of the ServantManager's possible responses is to throw a ForwardRequest exception, containing a new object reference. This response causes the POA to send LOCATION_FORWARD messages to remote clients, and to redirect local clients. Unfortunately, there can only be one ServantManager per POA.

It would be reasonably straightforward to adapt the migration scheme described in this chapter to work with the POA. The _wrap_home and _wrap_proxy objects would no longer be necessary. Each migrated object would still have an associated LifeCycleInfo object to register location changes, but there would no longer be one forwarding agent per object. The ServantManager would be responsible for all migrated objects, and would therefore have to maintain a mapping from object keys to the objects' current locations, rather that relying on the ORB's internal mapping from object keys to forwarding agents.

5.9 Summary

This chapter has detailed a design which permits CORBA clients to track objects as they migrate from location to location, in a way that is transparent to the clients. The system works by declaring an object's original location to be its *home* location. The home location is responsible for directing clients towards the object's current location. It is kept up-to-date with its object's current location via a LifeCycleInfo object.

Clients are directed to migrated objects using GIOP's LOCATION_FORWARD messages. Prior to the introduction of the POA specification, there was no standard interface to sending LOCATION_FORWARD messages. The implementation described in this chapter adds all of the ORB facilities required to support object migration to omniORB. The support is transparent to clients, and has a simple interface for implementations. It adds nearly no overhead to operation invocations. What little overhead there is only affects those objects which are designed to be able to migrate—on the server side, implementations may still derive from _sk_T rather than _lc_sk_T; on the client side, only objects which may migrate into the client's address space encounter any overhead.

5.9.1 Interoperability

Clients of life cycle objects should be able to use any ORB since only the standard GIOP location-forwarding mechanism is used. In practice, many ORB implementations fail to properly implement the standard[Har98], so clients may fail, especially when an object moves for a second time.

Although the server-side implementation described here makes use of a set of proprietary extensions to the CORBA model, the design could be applied to any ORB. It is very simple to move objects from a home location running omniORB to another ORB, by accessing omniLifeCycleInfo objects directly, rather than though wrapper objects. To implement a home location on another ORB without modifying its IDL compiler, the wrapper objects could be created manually.

6
Implementation Details

This chapter presents the implementations of some of the elements of DAWS, in order to illustrate the consequences of the framework and design. All parts of DAWS compile and run on both Solaris and Linux operating systems. It should be simple to port to any modern Unix, and, with the exception of the XDisplay, to Windows NT.

6.1 Virtual Display

Rather than creating a hardware-based display implementation, a simple 'thin' virtual display which runs under the X window system has been developed. It is roughly analogous in its operation to the VNC viewer, in that it provides a virtual frame buffer which is mirrored to the X display. None of the image rendering facilities provided by X are used, although it would be possible to create a different implementation which used far more of the facilities of the underlying window system. The advantages of the virtual frame buffer approach are that the implementation is very simple, and that much of the code could be re-used in a direct, hardware-based, non-virtual implementation.

An overview of the design of the XDisplay implementation is shown in figure 6.1. The implementation consists of the main `XDisplay` object, which provides the virtual frame buffer and interfaces with the X window system, and the implementations of three drawing Resources and two input Resources. The design is such that the vast majority of the code would work unchanged within a different host window system, or on a stand-alone display.

6.1.1 Drawing

The XDisplay requires a true-colour[1] X server with either 8, 15, 16, or 24 bits per pixel. Since the drawing Resources supported by the XDisplay all make use of 24 bits per

[1] i.e. displays with a fixed number of bits for each colour component, rather than a palette-mapped display.

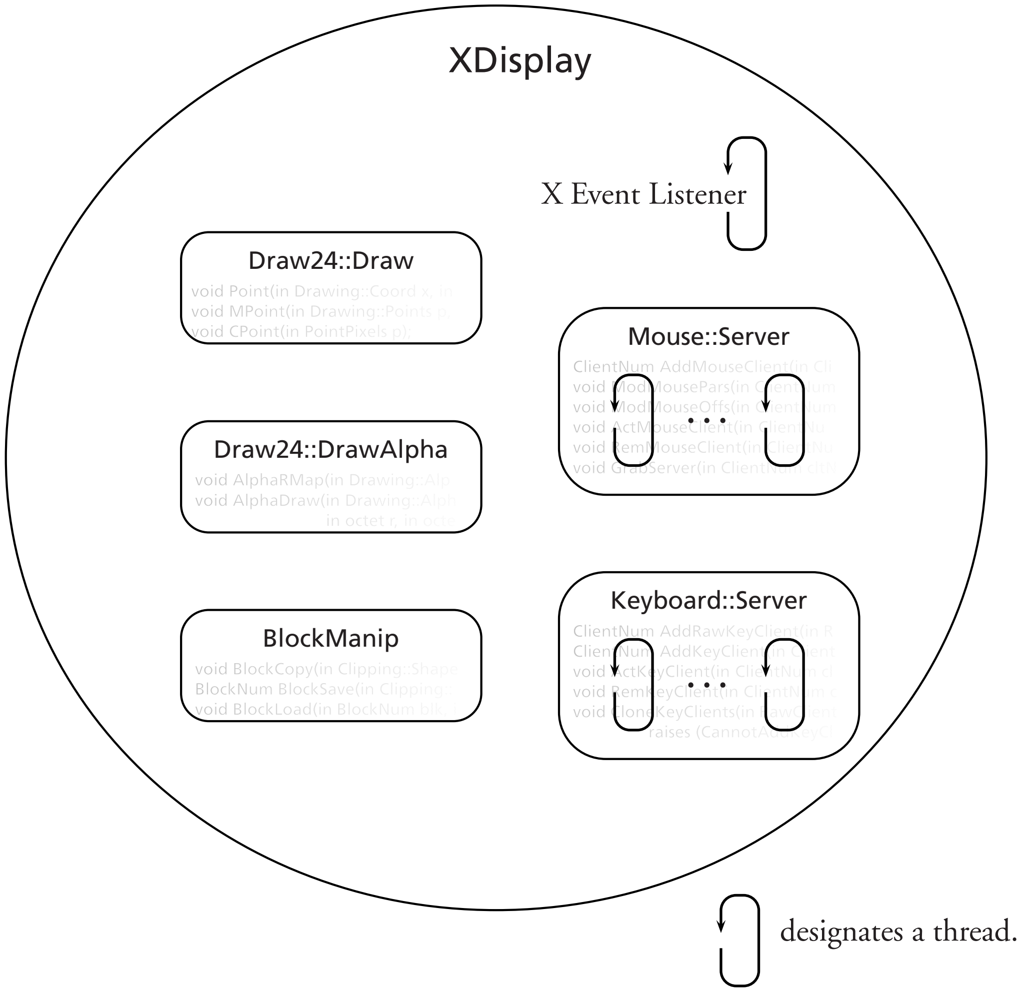


Figure 6.1: The design of the XDisplay virtual display

pixel, the implementations of all drawing operations possibly have to reduce the number of significant bits in the pixels they render. The rendering operations are all very simple to implement—they just set values within the virtual frame buffer according to the item they are drawing. Whenever a drawing operation has finished updating the virtual frame buffer, a function must be called to copy the contents of the frame buffer to the X server. This is performed synchronously, so that the display is guaranteed to have updated before the drawing operation returns to its caller. A faster turn-around to drawing clients could be achieved by performing X updates in a separate thread, but this would mean that the user's view would lag behind the rest of the window system, leading to a strange user interface experience.

A more efficient, although slightly more complex, implementation could use the X shared memory extension to represent the virtual frame buffer within the address spaces of both the X server and the XDisplay. In this way, updating the X frame buffer would involve a memory-to-frame-buffer copy within the X server, rather than a transfer from the XDisplay's address space to the X server via a stream protocol.

Multiple drawing operations can potentially be called from more than one client (or even the same client) simultaneously, resulting in several threads running within the XDisplay, all trying to write into the same virtual frame buffer. Rather than performing time-consuming thread control to mediate access to the frame buffer, the drawing implementations assume that the rest of DAWS prevents multiple objects attempting to render to the same part of the display. If two threads *do* attempt to render to the same part of the display simultaneously, the display could end in a confusing state. Imagine two threads rendering differently coloured blocks to the same part of the display at the same time. For each pixel plotted, the thread which plots it last will determine the colour displayed; due to the non-deterministic nature of thread scheduling, it is likely that one thread will come last for some pixels, but the other will come last for other pixels, resulting in a multi-coloured block.

In practice, the Layer system prevents independent applications from attempting to draw to the same part of the display at the same time, and well-behaved Components will only ever render within their stated shapes, so there should never be rendering clashes. It *is* possible, and indeed desirable, for multiple threads to be able to render to *different* parts of the display concurrently. A consequence of this is that the X library used by the XDisplay must be thread-safe, since multiple threads may instruct X to copy the virtual frame buffer to the X frame buffer simultaneously.

6.1.2 Input

The XDisplay has a single thread which receives all mouse and keyboard events from the X server. This thread converts the data to the event formats used by DAWS, and passes them on to the `Mouse::Server` and `Keyboard::Server` Resource implementations.

Every mouse event is passed to the mouse server Resource as the X and Y coordinates of the pointer, and the current state of the mouse buttons in the `Mouse::Buttons` format. The mouse server keeps a record of the previous button state, so that changes in state are properly noticed. For each client of the mouse server, there is a client handler sub-object with an associated thread within the server. Each of these sub-objects keeps a record of the types of events that its client has requested, the client's current position offset, and whether its client is currently active or not. When an event occurs, it is compared with the requirements of each client handler, and if an event should be sent to the client, the client handler's thread is triggered. This arrangement serves to de-couple the input handling from the responsiveness of the input clients.

The arrangements for keyboard input are almost identical. With keyboards, there is much less variation in client requirements. There are, however, two different client interfaces, for raw events and Unicode character events, so there are two classes of client handler object.

The client handler threads must cope gracefully if exceptions are thrown by the client operations they invoke. If a handler receives an OBJECT_NOT_EXIST exception, then it knows for certain that the client object no longer exists[2], and so the handler thread is stopped, and the handler thread is deleted. If a COMM_FAILURE exception is received, then the handler thread cannot tell whether the object still exists or not. In this situation, it waits for a short time before trying to send the event again (or a more recent event if more events have occurred). If it keeps receiving COMM_FAILUREs, it performs an exponential back-off before finally giving up (after five tries in the current implementation) and assuming the object to have ceased to exist. The handler may also receive a TRANSIENT exception due to the client object being in transit during a migration; in this case, omniORB's standard TRANSIENT exception handler automatically retries the invocation, with exponential back-off, until the object is available again, or a different exception is thrown.

6.1.3 Display migration

Migration of the XDisplay happens just as explained in section 3.5.5—upon receiving a call to its Move() operation, the main XDisplay implementation object clones itself at the new location. The newly created Display object (which is not necessarily an XDisplay) then triggers migration of all of the Resource objects.

During migration, the display's state must be transferred. It is not immediately obvious, however, how much state the display should be considered to have. The Display object itself maintains a list of DisplayInfo::Client objects which are to be informed of changes in the display, so that list must be transferred while migrating. Similarly, the two input servers must transfer their lists of clients. The various drawing Resources, on the other hand, share the same state—the contents of the virtual frame buffer. The obvious candidate for transferring this state is the base Draw24::Draw Resource.

Unfortunately, transferring the contents of the frame buffer is not a simple matter for two reasons. Firstly, all of the drawing operations use 24 bits per pixel, but if the host X server only supports lesser depths, the virtual frame buffer uses fewer bits per pixel. This situation could be avoided by always having a 24 bits per pixel buffer, as well as a buffer with the same depth as the X server, but that would make the XDisplay much less efficient. The second problem is that the destination display may be of a different size to the original. In this case, it is not possible to predict how the rest of the window system will re-configure the contents of the display to cope with the different size.

The solution is to consider the contents of the frame buffer to be a *cache* of the state of the Layers within the user's session. This means that, when the Display object

[2]Note that well-behaved input clients should unregister themselves from their input servers before they are removed, but the servers cannot rely on this.

migrates, it is not necessary for the contents of the frame buffer to be transferred. Once the migration has completed, all of the display's DisplayInfo::Clients are informed, and they are responsible for re-generating the contents of the display.

The XDisplay implementation does, in fact, have an option to transfer the frame buffer's contents upon migration. This was useful during testing, before all of the repainting mechanisms had been written.

6.2 Layers and Layer management

The implementations of the LayerMgr and LayoutLayer objects both have to perform a great deal of manipulation of Clipping::Shape structures. This section first details the implementation of the DAWSClip library which provides these shape manipulation functions, then goes on to describe other details of the LayoutLayer and LayerMgr implementations.

6.2.1 Shape manipulation

It would be sufficient for DAWS to provide functions to calculate the union of two shapes, the intersection of two shapes, and the inverse of a single shape, since all shape combinations which might be required could be constructed from sequences of these three functions. This would, however, be inefficient, since it would require multiple passes over the spans contained in the shapes, when a single pass is sufficient. In particular, when the LayerMgr clips the shape of one Layer against another, it must calculate both the visible shape of the lower Layer (i.e. the intersection of the lower shape with the inverse of the higher shape), and the union of the two shapes, so that the next Layer down may be clipped. It is possible to perform both operations in a single pass over the two original shapes.

The DAWSClip library provides the implementation of a SpanIterator object; the SpanIterator is initialised with two shapes, A and B, and possibly a bounding rectangle. It provides two member functions, NextYSpan() and NextXSpan(), which permit callers to iterate through all of the spans formed by the combination of A and B which fall within the bounding rectangle. Each span is classified by its extents, and three boolean flags: InA and InB indicate which of the original shapes the span is included in, and Last indicates that there are no more spans through which to iterate. As shown in figure 4.5 on page 70, each shape has its own origin within the coordinate system; the SpanIterator always specifies spans in global coordinates.

Figure 6.2 shows two simple rectangular shapes. A has a single Y span from 2–7 containing a single X span from 0–10; B has a Y span from 0–10 containing an X span from 2–8. The table in figure 6.3 shows the results of using the SpanIterator.

It is relatively simple to construct a new shape based upon the spans provided by

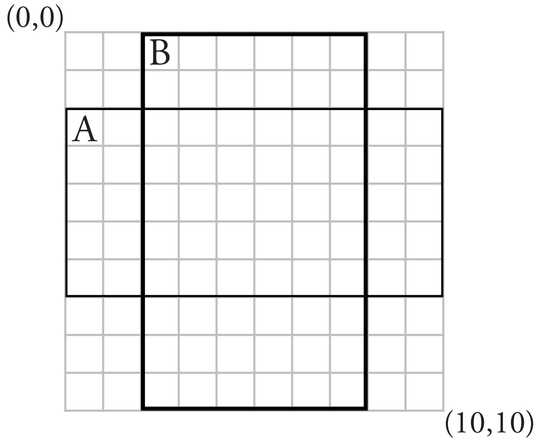


Figure 6.2: Two shapes to be merged

Call	Span	Flags
NextYSpan()	y: 0–2	InB
NextXSpan()	x: 2–8	InB Last
NextYSpan()	y: 2–7	InA InB
NextXSpan()	x: 0–2	InA
NextXSpan()	x: 2–8	InA InB
NextXSpan()	x: 8–10	InA Last
NextYSpan()	y: 7–10	InB Last
NextXSpan()	x: 2–8	InB Last

Figure 6.3: Iterating through the spans of the shapes in figure 6.2

the SpanIterator, with whatever combination of A and B is required. In the LayerMgr example, the clipped shape of B would contain the spans with InB set but InA not set; the combined shape would contain all spans with either InA, InB, or both set. The only remaining complication is that adjacent spans may have to be merged together, and empty Y spans have to be removed. So, for example, in the combined shape of A and B, the three X spans in the 2–7 Y span would be merged into a single X span.

Merging X spans is simply a case of seeing if the start point of a new span is the same as the end point of the previous span, and, if so, extending the previous span rather than adding a new one. Merging adjacent Y spans is a bit more complex, since it is not until all of the X spans in a given Y span have been processed that the Y span can be compared with the previous one[3]. When a new Y span is finished, its start point is compared with the end point of the previous span. If they are the same, then the X spans of the two Y spans are compared in turn. If *they* all match, then the new Y span is deleted, and the previous one is extended. Creating a new shape from the output

[3] It would be possible to compare the X spans of two adjacent Y spans as they were added, but this would conflict with the merging of the X spans.

of a SpanIterator may therefore require many passes over the contents of a particular Y span, but such situations are rare.

The SpanIterator itself works by maintaining which span from each of the original shapes it should look at next, and whether it is currently inside or outside those spans. This means that, at any given time, there is a 'next point of interest' for each shape. When the next span is required, it takes its start point from the previous span's end point, and its end point from the minimum of the two shapes' points of interest. If the SpanIterator has a bounding rectangle then, when it is initialised, it finds the first Y span which falls at least partially within the rectangle using a binary search. Each time a new Y span is chosen, the first X span within the bounding rectangle is found, also using a binary search.

The DAWSClip library provides a number of functions which combine shapes using the output of the SpanIterator, to further simplify programs which perform common shape manipulation operations, such as unions and intersections.

6.2.2 LayerMgr implementation

The LayerMgr implementation is responsible for maintaining an ordered list of Layers, where the order represents the top-to-bottom stacking of Layers, and recording the current geometry and natural shape of each Layer. This information is stored in a linked list.

Whenever the LayerMgr receives a request to change a Layer's geometry, shape, or position within the stack, it must recalculate the visible shapes of each of the Layers, and inform any Layers whose visible shapes have changed. It informs Layers though their ClipChange() operations. ClipChange() not only informs the Layer of its new visible shape, but also includes a 'dirty' shape which is the region which must be repainted.

It is possible for the LayerMgr to receive multiple concurrent invocations to operations which modify Layers' states, but such situations are uncommon since changes are usually in response to interaction from the user. The LayerMgr therefore simply has a single mutex to guard all of its state, rather than utilising a more complex scheme to achieve greater concurrency. Operations which modify the parameters of a Layer tend to require that the visible shapes of the Layer in question and all of the Layers 'beneath' it are recalculated. This recalculation is performed in a separate thread so that the caller of the operation may continue as soon as possible. If the caller tries to call make another invocation, it will find itself blocked until the recalculation has finished, since the recalculation thread locks the global mutex.

Concurrency can be very important when it comes to communicating with the individual Layer objects. Before the Layers' visible shapes may be recalculated, each Layer object's SuspendOutput() operation must be called; once all of the visible shapes are known, the Layers must be re-activated, either by calling ResumeOutput() (if the Layer's shape has not changed), or by calling ClipChange() (if the shape has changed).

Rather than invoking operations on the Layer objects sequentially, the LayerMgr maintains a separate thread for each Layer. The main recalculation thread triggers the Layer handling threads when appropriate. As was explained in connection with Component Layout in section 4.7.1, invoking operations on different Layer objects in a concurrent fashion is only productive if there is some genuine concurrency available to be exploited. There is a performance gain if the objects are running on different machines on the network, or if they are running on a single multi-processor machine (or some combination of the two), but there is a performance *loss* if all of the objects are running on one single processor machine. The LayerMgr implementation does not attempt to vary its threading behaviour according to the locations of Layers; the consequences of this are evaluated in section 7.1.4.

When the LayerMgr is recalculating the Layers' visible shapes, it compares each Layer's new shape with its old shape. There are five possible outcomes of this comparison:

1. The visible shape has not changed in any way.

   In this case, output from the Layer is immediately resumed with a call to ResumeOutput() by the Layer's handling thread.

2. The Layer has moved, but its visible shape is still the same.

   In this situation, if the Display object supports the BlockManip Resource, the LayerMgr copies the Layer's imagery from the old location to the new location. It then calls the Layer's ClipChange() operation with an empty dirty shape. If the Display does not have a BlockManip Resource, then ClipChange() is called with the dirty shape set to the entire visible shape. This is much less efficient since the Layer must repaint the entire visible area.

3. The visible shape has changed, but the Layer's position and size has not.

   ClipChange() is called with a dirty shape corresponding to those regions which are now visible but were not before.

4. The visible shape and the position have changed, but the size has not.

   If there are any regions which were previously visible that are still visible, and the display has a BlockManip Resource, the unchanged regions are copied, then ClipChange() is called with the dirty shape set to any regions which are newly visible. Otherwise, the Layer's ClipChange() operation is called with the dirty shape set to the entire visible shape.

5. The size of the Layer has changed.

   The Layer will have to recalculate its layout to fit the new size, so ClipChange() is called with a dirty shape equal to the visible shape.

The use of the BlockManip Resource is a good example of the extensible nature of DAWS. If the display supports the Resource, the LayerMgr uses it to gain performance; if the Resource is not supported, the LayerMgr is still able to operate.

6.2.3 LayoutLayer implementation

The implementation of the LayoutLayer has three main responsibilities: it must provide clipped drawing Resources, it must manage virtual input Resources, and it must interact with a LayoutMgr object.

The current LayoutLayer implementation provides the Draw24::Draw and Draw24::DrawAlpha drawing Resources, although it only makes the DrawAlpha Resource available if the host Display object also supports it. The drawing operations use a combination of functions provided by the DAWSClip library and direct manipulation of shapes to make sure that they only allow pixels to be rendered within their current visible shapes.

Some of the drawing operations, for example those which plot a number of independent points, have an obvious implementation in which the points which fall outside of the visible shape are removed, and the remainder are passed on to the same operation within the display's drawing Resource, with appropriate translation for the Layer's position. Other operations, such as those which render solid rectangles or rectangular maps, have two possible implementations, depending upon the Resources provided by the display. In both cases, a DAWSClip library function is used to establish the visible shape of the rectangle which is to be rendered. The resulting shape will not, in general, be rectangular, so it will take multiple calls to operations of the display's Draw24::Draw Resource to render it. If, however, the display also supports the Draw24::DrawShape Resource, the complete shape can be rendered with a single operation invocation, thus reducing the effect of network latency. Clipped drawing therefore provides another example of how the functionality and performance of DAWS can be extended through the Resource mechanism.

Implementation of the virtual input Resources is very simple, since the vast majority of the work is actually performed by the real input Resources running on (or along-side) the display. The virtual Resources just maintain a mapping between their own clients and the actual input Resources' view of those clients, and activate and deactivate clients as necessary. The virtual mouse server also has to inform the real mouse server of the Layer's location so that coordinates are reported correctly.

Thread control comes into play again when it comes to interacting with the LayoutMgr. Whenever the Layer's ClipChange() operation is called (by the LayerMgr), the clipping region used by the drawing operations must be changed, and the LayoutMgr must be instructed to repaint the dirty areas. Many drawing operations may be invoked simultaneously by multiple Components within the Layer, so it is not appropriate to use a simple mutex to guard the state of the visible shape. A multiple reader, single writer interlock is used instead[4].

It may take a significant time for the Components, under control of the LayoutMgr, to repaint themselves when the Layer's visible shape has changed (signalled by a call to the Layer's `ClipChange()` operation). To increase performance, there is a separate thread within the Layer which is in charge of calling the LayoutMgr's `Paint()` operation, so the `ClipChange()` operation can return to the caller (the LayerMgr) before the repainting has completed. This means that it is possible for `ClipChange()` to be called *again* before the previous repaint has finished, which is particularly likely when windows are being dragged. To avoid problems with this, when the repainting thread is triggered it makes a local copy of the current dirty shape, then clears the dirty shape. Upon receiving a call to `ClipChange()`, the Layer modifies its record of its dirty shape according to the new visible region and specified dirty region. Once the repainting thread has finished, it checks to see if there is a new dirty shape and, if so, immediately starts another repaint. In this way, multiple calls to `ClipChange()` are aggregated into as many repaints as the system is able to perform, but there is never a build-up of pending repaints.

To summarise, the implementations of the LayerMgr and LayoutLayer objects both demonstrate the extensibility that DAWS achieves through its Resource mechanism. They provide multiple implementations of the same functionality so that when advanced Resources are available, they are used for increased performance, but when such Resources are not available, less advanced Resources are used instead.

6.3 Text rendering

The text rendering implementation provides a useful illustration of how existing code can be integrated into DAWS. Rather than designing a new font format, or implementing a new renderer for an existing format, DAWS uses the FreeType[Tur98] library for rendering TrueType[App97] fonts.

The FreeType library is written in C, and is released under an open-source license. It provides the lowest levels of a font rendering system, by supplying functions which render individual glyphs from a font, and an interface to metric (i.e. width and height) and kerning information. It does not support caching of previously-rendered glyphs, or any means to render whole strings of glyphs. The `TextRender` implementation sup-

[4]Multiple Components simultaneously calling the drawing operations need to read the visible shape without blocking each other; the `ClipChange()` operation, called by the LayerMgr, needs to write to the shape, and must block out the drawing operations while it does so.

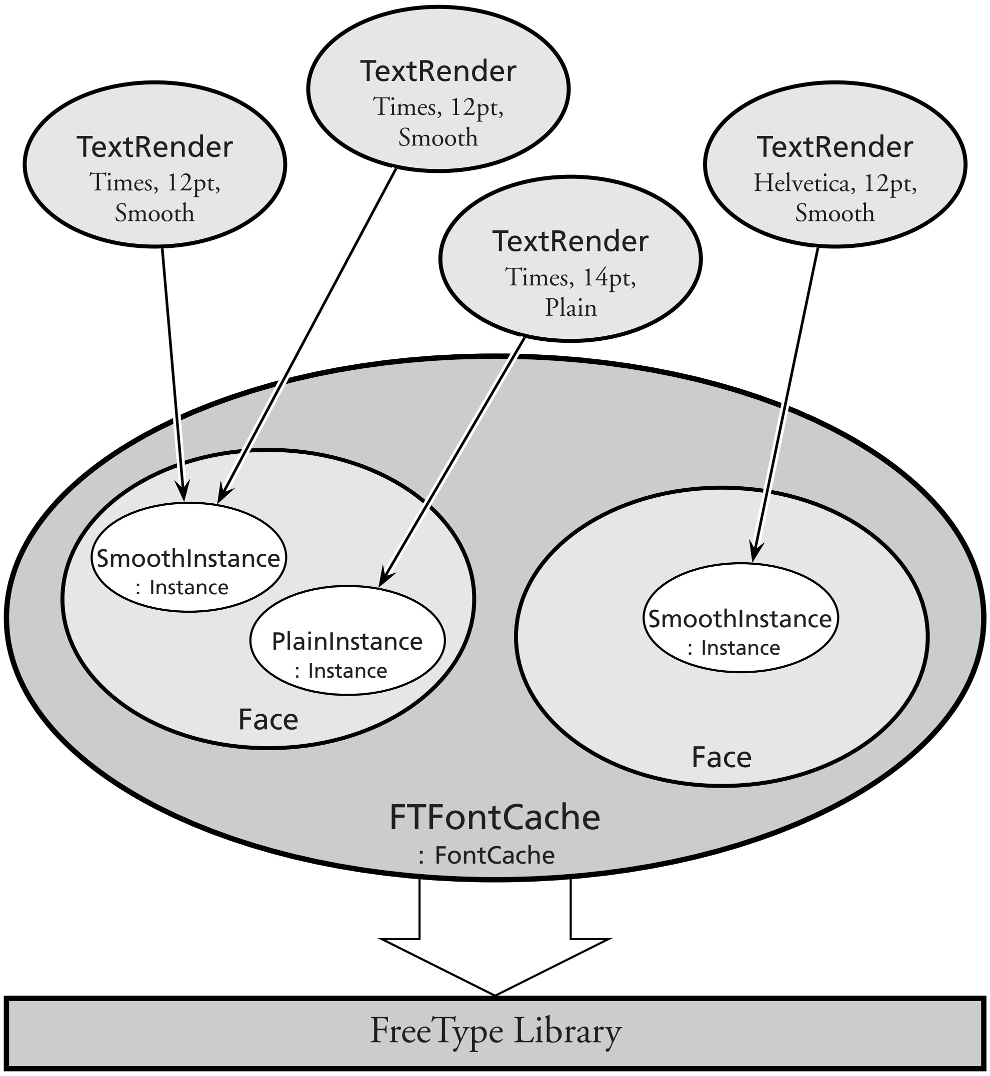


Figure 6.4: Objects used in text rendering

plies these facilities.

It would be possible for each `TextRender` object to provide its own cache of rendered glyphs but, as explained in section 4.8.3, each `TextRender` draws into a single ResourceFinder's drawing interfaces. Since it is common for different clients to wish to render the same font into many different Layers, this would lead to replication of the same cached glyphs. That would be inefficient both in terms of storage, and in processing time spent rendering previously-rendered glyphs.

Instead of this, DAWS uses the arrangement of C++ objects shown in figure 6.4. The `FTFontCache` object provides an abstraction around the underlying FreeType li-

Figure 6.5: Some text rendered with and without hinting

brary. The FTFontCache class is derived from the abstract FontCache class, so the TextRender implementation code can be re-used with other font rendering engines. The FTFontCache is responsible for initialising the FreeType library, and managing the list of available fonts.

When the TextRenderFactory is asked to create a new TextRender object, it first acquires a Face object by a call to the FTFontCache's GetFace() function. There is a single Face object per TrueType font in use. The Face object is responsible for calling FreeType functions to deal with the TrueType font file, and for managing the different TrueType character encodings. TrueType basically uses the Unicode[Uni96] character encoding (as does DAWS), but most fonts only contain a small subset of the Unicode characters. The Face object interfaces with FreeType's character mapping functions.

Once the TextRenderFactory has a Face object, it calls its GetInstance() function to acquire an Instance object with the required properties. Properties include the size of the font, whether it is hinted, and whether it is anti-aliased. If an Instance with the requested properties already exists, the Face returns a pointer to the existing object, rather than creating a new one. The Instance objects are responsible for caching rendered glyphs, and for creating DAWS structures (RectMaps, Alpha maps, and one bit-per-pixel Masks) containing strings of cached glyphs. When rendering a non-smoothed glyph, the FreeType library uses a one bit-per-pixel image format; anti-aliased glyphs are rendered into an eight bits-per-pixel image. The two sub-classes of the Instance class, PlainInstance and SmoothInstance, deal with these differences.

The Instances are also responsible for calculating metric information and reporting it to the TextRender objects. This is done on a per-instance basis, rather than per-face, since hinting can cause glyphs to have different relative widths at different sizes. Figure 6.5 shows a font rendered at 12 points, both with and without hinting. Not only is the hinted version far clearer, but it also significantly wider. This scheme for calculating metrics is ideal for on-screen presentation, but problematic for WYSIWYG systems where the on-screen representation is supposed to closely resemble the printed version. In these circumstances, it is better to evaluate font metrics in a resolution-independent manner. The TextRender interface and implementation does not provide any support for this, since a WYSIWYG application (or more usefully a set of WYSIWYG Resources) would have to have detailed knowledge of text rendering issues in order to manage printing, so it may as well perform its own on-screen text rendering too.

The Instance and Face objects implement a reference counting mechanism so that they may be removed when there is no longer a need for them. Unfortunately, since the TextRender objects are CORBA objects, there is no way of telling when they have no clients and can be safely removed. It would be possible to implement a client pinging system, but that would place an unreasonable burden on the clients. Instead, the TextRenderFactory periodically scans each of the TextRenders it has created, and attempts to contact the ResourceFinders they are set to draw into. If a TextRender's ResourceFinder no longer exists then the TextRender can no longer operate, and it may be safely removed. Text rendering is generally carried out on a relatively capable machine, so there is no need to remove abandoned TextRenders particularly rapidly. The current implementation only performs a scan once per hour.

The remaining problem with object removal is that it is possible to use a TextRender to return pixel maps containing rendered text, rather than to render directly into a ResourceFinder. Clients which wish to use TextRenders in this way should either cope gracefully with the unexpected disappearance of an object, or should create their own dummy ResourceFinder which the TextRenderFactory can ping.

Like most DAWS objects, TextRenders can migrate. Implementing the migration is essentially very simple: the only items of state owned by a TextRender are its properties, and the ResourceFinder it is currently rendering into. Not all TextRenderFactory instances necessarily support the same fonts as each other, however, so cloning an existing TextRender at a new location may fail (with a CannotCreate exception) if the requested font or font properties are not available. In this situation, the TextRender object would not be able to migrate to that location, so it would have to try a different TextRenderFactory or remain where it was.

Each TextRenderFactory must know which font files are available to it, and what names and properties they have. Since new fonts can be installed, and existing fonts can be removed, this information may change during the life-time of the factory. Rather than trying to scan for font files, the system uses a simple configuration file which provides the required data. The format of the configuration file is shown in Appendix B. When this file is changed, the TextRenderFactory must be informed so that it can update itself. If fonts are to be removed, the font files themselves must not be removed until the factory has been given the updated configuration file, and it has confirmed that it is safe for the files to be deleted. If a font file is currently in use, it should not be deleted[5].

In summary, the TextRender implementation demonstrates a number of techniques

[5]Standard Unix file handling should mean that if a file is deleted when a process has an open file handle to it, only the file's directory entry is deleted. The file itself should not be deleted until all open file handles to it have been closed. Unfortunately, NFS-mounted network file systems do not have this property. Relying on this behaviour would also limit the TextRender implementation's portability.

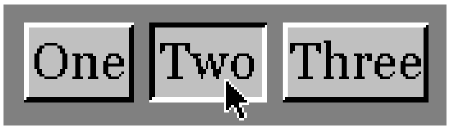


Figure 6.6: Button Components

which are generally useful. It shows how an existing library can be wrapped in a Layer of CORBA objects, thus making it available to DAWS. It also shows that there can be benefits to creating very 'thin' CORBA objects which delegate most of their processing to other C++ objects. This permits multiple CORBA objects to share cached state.

> Most importantly, the mechanism used to clean up abandoned `TextRender` objects demonstrates that although the lack of distributed garbage collection in CORBA places a greater burden on the programmer, it makes it possible to design efficient collection schemes which make use of knowledge about the system as a whole.

6.4 Component implementations

As explained in section 4.7, the DAWS design does not include the designs of *any* user interface Components at all. This section outlines the designs and implementations of a number of example Components which have been created in order to test the system. All of the Components, like most DAWS objects, may migrate. With the majority of Components, this is simply a case of sending the argument and contents sequences, and a small amount of other state, so the details are not described here.

6.4.1 Buttons

The Button Component provides a simple push button with a text label. The user is considered to have clicked the Button when they click and release the mouse button over the Component's area. The pointer is permitted to stray outside the Component while the mouse button is still pressed, as long as the pointer returns to the Component before the mouse button is released. If the mouse button is released when the pointer is not over the Button, a click is not registered; similarly, if the mouse button is pressed before the pointer is over the Button, then released over the Button, a click is not registered. The Button's imagery changes to look 'pressed' during a click operation. Figure 6.6 shows three buttons, the centre of which is being clicked on.

Like most Component interfaces, the `C_Button` interface simply derives from the `Component` interface, without declaring any extra operations. Since the Button needs

Name	Type	Purpose
Mandatory arguments:		
`label`	Text::ustring	Label to put on the button
`font`	TextRender	Font with which to render label
Optional arguments:		
`hicol`	Draw24::Pixel	Colour of highlights.
`locol`	Draw24::Pixel	Colour of lowlights.
`labelcol`	Draw24::Pixel	Colour of label text.
`bgcol`	Draw24::Pixel	Colour of button background.
`srange`	Size::Range	Size range in addition to label size.
`help`	Text::ustring	Help string.
`stateto`	EventHandler::Spec	Destination for full state information. Boolean state value is true for pressed, false for unpressed.
`clickto`	EventHandler::Spec	Destination for click information. Boolean state value is true when a successful click is made; false is never sent.

Figure 6.7: Arguments accepted by the Button Component

to receive mouse input, it also multiply-inherits the Mouse::Client interface. Comments in the IDL file declare the arguments shown in figure 6.7. Most of the arguments have default values, and so are optional. The `label` argument is clearly necessary, and the Button must know which font it should use to render the label. Rather than specifying the required font properties to the Component, a reference to a TextRender object must be specified. This permits many Components to share the same TextRender object, and also means that the Button implementation does not have to know anything about how font properties are specified.

The implementation of the C_Button object is extremely simple. When it is initialised, it makes sure that no sub-Components have been specified for it, and then processes its arguments. This is simply a case of performing a sequence of string comparisons of the argument keys, for each of the valid arguments. When a valid argument is found, its value is extracted from the argument's any into the relevant type. If an argument value is of the wrong type, an unknown argument key is given, or any of the mandatory arguments are not given, an ArgError exception is thrown. This argument processing stage is essentially the same for all Components which accept arguments.

Once the arguments have been processed, the Button works out its size range based

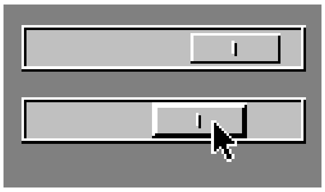


Figure 6.8: Horizontal scroll bars

upon the specified label text and font. If the srange argument was given then the specified range is added to the size of the label; otherwise, some default values are added to form the minimum, base, and maximum sizes of the button. The final initialisation step is for the Button to register itself with the LayoutMgr's MouseMgr Resource, so that it receives mouse input. The MouseMgr grabs the mouse server on the Button's behalf when the user clicks within the Button's shape.

The Button implementation is not an active object—it has no internal threads—so it only acts upon operation invocations. When it is instructed to paint itself, it draws the button imagery with a number of rectangular blocks, and a call to the TextRender object. If the ResourceFinder (almost certainly actually a LayoutLayer) that it is rendering into supports the Draw24::DrawCBlock Resource, then the whole button (excepting the label) can be rendered with a single call to CBlock(). Otherwise, a significant number of individual calls to Block() are necessary (nine with the imagery shown in figure 6.6).

When the Button receives mouse data, it checks the button state. If the left mouse button is pressed, then the Button's imagery is updated to indicate whether the pointer is inside or outside the Button's shape. If a stateto argument was specified, then events are sent to the EventHandler to indicate changes in the state. Updating the Button's imagery is simply a case of changing the button's border, so the label does not need to be re-rendered. If the mouse button is released when the pointer is inside the Button's shape, and the Button was previously pressed down, then the Button is deemed to have been clicked. If a clickto argument was specified, then an event is sent.

Both the state change and click events are sent synchronously, within the MouseData() operation, before any changes in the Button's imagery are made. This means that the user receives visual feedback of the latency involved in sending the event.

6.4.2 Scroll bars

There are two separate scroll bar Component interfaces, C_HScrollBar and C_VScrollBar, for horizontal and vertical scroll bars respectively. Their implementations are essentially identical, so they are discussed together. As with the Button Component, the scroll bar interfaces are derived from the Component and Mouse::Client interfaces, and contain no extra operations. They are also derived from the EventHandler interface, so that various aspects of their state may be updated.

Each scroll bar has a number of properties which determine both its visual appearance and its behaviour. The scroll bar position is represented as a CORBA long (i.e. signed 32 bits) with upper and lower bounds. The position of the scroll bar's handle (the part which the user drags) represents the scroll bar's current value within the permitted range. The scroll bar handle has a notional 'window size'—the length of the handle represents the proportion of the scroll bar's range covered by the window. So, if the scroll bar's range is from 0 to 10, the window size is 4, and the scroll bar is 100 pixels in length, then the handle will be 40 pixels long. The handle is never permitted to get shorter than a minimum length, however. If the user clicks inside the scroll bar, but outside the handle, the value is incremented by the window size. The scroll bar does not have arrows at its ends—separate Components are used for that—but it does have another property value to indicate by how much its value should be incremented or decremented when arrows are clicked.

The above properties are set in initialisation arguments named `minval` and `maxval`, which are mandatory, and `window` and `step`, which default to a value of 1. The Component also accepts events with the same names to change the values during the scroll bar's life time. The scroll bar's value can be set using the `value` argument and event; the value defaults to the minimum value. The boolean events `valueup` and `valuedown` respectively increment and decrement the scroll bar's value by the step. Two EventHandler::Spec arguments, `stateto` and `releaseto`, permit the scroll bar's value to be sent to an EventHandler: `stateto` sends updates for each movement of the handle; `releaseto` only sends an event when the mouse button has been released after changing the value. Both types of event are also sent in response to the receipt of a `valueup` or `valuedown` event. The required size range of the scroll bar is specified with the `srange` argument. Finally, the Component also accepts the same colour and help string arguments that are accepted by the Button.

The scroll bar implementation is relatively uninteresting, since it is very similar to the Button. Unlike the Button implementation, however, the scroll bar implementation *does* use a separate thread to send events, since the most important aspect of moving the scroll bar handle is its final destination. If events were sent synchronously with the MouseData() operation, update of the handle's position could lag quite significantly behind the mouse pointer.

6.4.3 Horizontal and vertical boxes

A small number of other low-level Components, including boxes for text input and output, have also been developed, but there is not much of interest in their implementations. To build useful user interfaces it is, of course, necessary to arrange collections of these low-level Components into sensible patterns. To this end, horizontal and vertical boxes, loosely based upon the boxes provided by the TEX typesetting system, have been implemented.

An HBox arranges its contents in a horizontal line, and stretches them vertically so they all have the same height. Its horizontal size range is the sum of the horizontal size ranges of its contents. When the HBox is stretched to be wider or thinner than its base width, the widths of its constituent Components are set in proportion to how much they have contributed to the overall stretchiness of the HBox. The box's minimum height is the largest of the minimum heights of its contents; the maximum is the smallest of its contents' maximum heights. Its base height is the mean of the base heights of its contents, as long as that is within its size range; if it is outside the range, the base height is the closest value to the mean which *is* in the range. A negative base height for a Component indicates that the Component does not have any preference as to what size within its range it takes. This makes it easy to construct Components which expand to fill the available height, but do not influence the height. The VBox performs the equivalent Component arrangement in a vertical direction.

From this description it is clear that it is possible (and indeed likely) for an HBox to be unable to form due to lack of vertical stretch in its contents (and for Components in a VBox to have insufficient horizontal stretch). As described, it would also be the case that the Components all 'touched' each other, with no space between them. Both of these situations are resolved through the use of *glue*. Glue is inserted between each of a box's constituent Components, and also at the start and end. Glue has a minimum, base, and maximum size, in the direction of the box—glue in an HBox stretches horizontally; glue in a VBox stretches vertically. Glue within a box is specified in the box Component's arguments with a sequence of Size::Glue structures:

```
module Size {
  struct Glue {
    Drawing::Coord min;
    Drawing::Coord base;
    Drawing::Coord max;
  };
  typedef sequence <Glue> Glues;
};
```

The sequence must contain one Glue structure for each possible glue position in order, even if no glue is required. In that case, the Glue structure should contain all zeros. Since it is a common case for all of the glue between Components to be identical, the short-cut of specifying only three Glue structures in the sequence is permitted, in which case the first and third Glues are used to specify the glue at the start and end of the box, and the second Glue specifies all of the inter-Component glue.

In situations where a particular Component within an HBox does not have enough vertical stretch to fit with the other Components, it can be put in a VBox with suitable glue surrounding it. Similarly, HBoxes within VBoxes can be used for Components with insufficient horizontal stretch. An example of glue in action may be found in section 6.5.

The HBox and VBox implementations are quite simple—they just calculate their size ranges from the sizes of the Components and glue they contain, and then arrange their contents based upon the size to which they are set. As mentioned in section 4.7.1, it is not obvious whether it is best for the boxes to inform their contents of their geometries in parallel, using separate threads, or to inform them one after the other. Section 7.1.4 evaluates the two approaches.

6.4.4 Summary

This section has detailed the designs and implementations of a number of user interface Components. The ease with which Components are created means that it is simple for programmers to create new Components, both to extend the basic capabilities of the window system, and for more specialist purposes. Applications with any sort of user interface complexity benefit from the creation of application-specific Components, since that reduces the application core's dependencies on the particular user interface arrangement. It is also very easy to replace Component implementations to create special-purpose versions, or simply to change their visual appearances. The semantics of particular Components, both at a programming and a user interaction level should not be changed, however.

6.5 User interface description

Although creating and initialising Components is conceptually very simple, the C++ code it requires is extremely inconvenient. Components form a tree, but the code to create the tree is linear, and must start with the leaf Components. Initialising each Component involves creating the Component object with a factory found through the FactoryFinder, then creating argument and content sequences and calling the object's initialisation operation. These steps involve a significant number of lines of code. The code can be simplified to some extent with the use of pre-processor macros, but the result is still confusing, and it is very difficult to visualise the Component tree by looking at the program.

DAWS solves this by providing a very simple interpreted GUI description language named *Quill*, the Quick User Interface Layout Language. Quill is an unusual language in that the majority of the 'keywords' used to describe the user interface are the names of Components, and are thus not known to the language at all, since the set of Components that may be used is unlimited. Quill does understand a small number of keywords which are used to specify argument types. Figure 6.9 shows an example Quill description; figure 6.10 shows the user interface generated from the description. The full grammar of Quill is given in Appendix C.

A Quill description consists of a single *expression* in a form designed to be very sim-

```
Border (width = 2, linecol = col(0xffffff), pad = glue(10)) {

  VBox (glue = glue(0, 5 10 15, 0)) {

    HBox (glue = glue(0, 5 10 15, 0)) {
      Button (font=tr, label="One",    clickto=ehandler(app, one))
      Button (font=tr, label="Two",    clickto=ehandler(app, two))
      Button (font=tr, label="Three",  clickto=ehandler(app, three))
    }
    Block (srange = size(—, 2), col=col(255, 255, 255))

    HScrollBar <sb> (minval=1, maxval=10, window=3, value=6,
                     srange  = size(—, 14 18 22),
                     stateto = ehandler(app, scroll))

    HBox (glue = glue(10 20 30, 5 10 15, 10 20 30)) {
      Button (font=tr, label="—", clickto=ehandler(sb, valuedown))
      Button (font=tr, label="+", clickto=ehandler(sb, valueup))
    }
  }
}
```

Figure 6.9: Quill description of a user interface

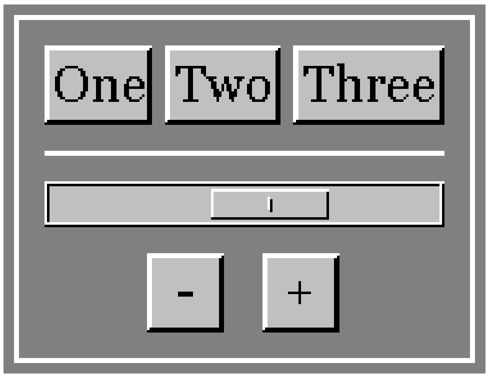


Figure 6.10: The user interface generated by the Quill description in figure 6.9

ple to parse. An expression consists of a Component type name followed by an optional instance name enclosed in angle brackets, followed by an optional comma-separated list of arguments within parentheses, followed by an optional list of content expressions, surrounded by curly brackets. Arguments take the form '*argname*=*argvalue*', where the *argname* is an argument name declared within the Component's interface, and the *argvalue* has a type recognised by Quill.

Quill currently recognises three simple argument types, and four complex types. The simple types are `CORBA::Long`s which are specified just as the decimal number or as a hexadecimal number of the form '0x*nnnn*', Unicode strings surrounded by '"' characters[6], and object references represented by a text name. Object reference names are resolved either from Component names (in angle brackets) declared within the same Quill description, or from references registered by the application. The `font` arguments of the buttons declared in figure 6.9 use the `TextRender` object reference named 'tr', registered by the application.

The complex argument types are used to place structures, rather than basic IDL types, into the `any`s in the Component's argument sequence. An *argvalue* of 'col(*r*,*g*,*b*)' creates a `Draw24::Pixel` structure with the specified red-green-blue components; alternatively, 'col(0x*rrggbb*)' specifies the RGB triplet using the common hexadecimal notation. An *argvalue* of 'ehandler(*handler*, *key*)' creates an `EventHandler::Spec` structure with the specified event handler and event key. The `EventHandler` object reference is resolved in the same way as references specified directly as the *argvalue*. In the example in figure 6.9, events from the top three buttons and the scroll bar are sent to an `EventHandler` registered by the application with the name 'app', and events from the lower two buttons are sent to the scroll bar Component, registered with the name 'sb'.

The 'size(*w*,*h*)' *argvalue* creates a `Size::Range` structure. The width and height specifications can have several forms: the full form of '*min base max*' explicitly provides the minimum, base, and maximum lengths; the abbreviated '*base*' form sets all three values to be the same, thus specifying a length with no stretch; finally, a specification containing a single '–' character sets the minimum size to zero, the base size negative, and the maximum size the largest possible coordinate value. This form creates a Component which stretches to fill all of the available space, without influencing the size of its container. The final complex argument takes the form 'glue($glue_1$, $glue_2$, …, $glue_n$)', and creates a sequence of `Size::Glue` structures. As with the specification of size ranges, each glue specification can either contain a triple declaring the minimum, base, and maximum size, or it can contain a single value to create glue with no stretch.

6.5.1 Quill interpreter implementation

A Quill user interface description is interpreted into a tree of Components in three phases. In the first phase, the Quill expressions are parsed (using a lex/yacc parser) to create a tree of expressions. At the same time, the argument sequence for each Component is created and initialised as far as possible. Argument values which refer to other Components in some way can not be fully initialised at this stage, since the Component objects have not been created yet.

[6] To be able to fully specify Unicode strings, the Quill description would have to be stored as Unicode. The current implementation actually uses the ISO Latin 1 character set, so only those characters are available within strings.

In the second phase, all of the Component objects are created with factories found through the FactoryFinder. Component creation could, in fact, be performed during the first phase, but it is not, since if there are any errors in the Quill description, Components which had just been created would have to be removed. It is better to be sure that the whole Quill description is valid before any Components are created. At this stage, the names of any named Components are added to the name mapping.

In the third and final phase, any argument sequences which were not previously fully initialised are completed. Then, each Component's `Initialise()` operation is called in a depth-first traversal of the tree. At this stage, the Components may throw exceptions to indicate errors within their arguments or their sub-Components, in which case all of the Components must be removed, and an error returned to the application.

6.5.2 Other approaches

Quill is purposefully an extremely simple language. The parser for it is very small, so it can easily be included within an application, either directly, or as a shared library. The difficulty with Quill is that the available argument types—size ranges, event handlers, and so on—must be hard-coded. It is not difficult to add new types, but Quill's extensibility is slightly restricted.

A more extensible language implementation could be built by permitting parts of the parser to run remotely, using objects found with the FactoryFinder (or the Naming service). The main parser would pass the text of a part it did not understand to a suitable object, which would return the necessary structure within an `any`. In this way, it would be possible to add new argument types without having to modify existing instances of the parser.

The present version of Quill is interpreted, and so involves a run-time overhead. This overhead is quite small, but it could be considered a problem in some circumstances. An alternative would be to create a Quill compiler which emitted C++ code (or some other language) which created the necessary Components. Unfortunately, since Quill does not know about the types of Components it creates, a compiler would not be able to check that the Quill description was valid; such checking would still have to be performed at run-time. Compiling Quill would also slow down GUI development since changes to the Quill description would only take effect after lengthy recompilations.

To facilitate interaction with other systems, it would possibly be worth creating a user interface description language based upon XML[W3C98]. Such a language could possibly be used to specify user interfaces for non-graphical systems as well as window systems like DAWS, but it would tend to be less readable than a Quill description.

A completely different approach to user interface description would be to create a direct-manipulation design application which permitted the user to create a user interface in a graphical manner. There are two considerable difficulties with this. First, the visual appearance of DAWS Components is not fixed, and aspects of a user interface such as fonts and colours are configurable by the user. This means that a visual design system would tend to create user interfaces which only looked acceptable with a narrow range of Component options, since users would be likely to try to arrange Components visually, rather than based upon some constraint specification. The second problem is that the current DAWS system has no mechanism for querying what Components are available, and specifying Component arguments in a machine-intelligible manner is an extremely hard problem. Any design application would therefore probably be quite constrained in its extensibility.

6.6 Summary

This chapter has shown that, by splitting the window system into a large number of reasonably fine-grained objects, the implementations of individual objects become quite simple and flexible.

- The most significant difficulty with implementing DAWS objects is that the design requires a great deal of multi-threading; different threading schemes must be used in different situations.
- The use of the Resource mechanism to provide flexibility and extensibility has been demonstrated. Many of the implementations can make use of particular Resources to gain performance, but automatically fall-back to using more common Resources if necessary.
- The TextRender implementation demonstrates how the lack of built-in reference counting and garbage collection in CORBA means that very light-weight object removal schemes can be created based upon wider knowledge of the system.
- The Quill GUI description language is a good example of how small application specific languages can greatly simplify awkward programming problems.

7
Evaluation and Discussion

This chapter evaluates how well DAWS meets the requirements it was designed to address, and introduces some simple performance measurements. Next, it discusses some ways in which applications could be designed so as to benefit from DAWS as much as possible. It goes on to describe some situations in which DAWS' highly adaptive nature would be beneficial, and finally outlines a way in which the DAWS approach could be integrated with existing window systems.

7.1 Window system performance

It is very difficult to measure the performance of a window system in a meaningful way. The user's perception of the interface with which they are interacting depends greatly upon their expectation of the system's performance. It is a general rule that the system is too slow if the user becomes frustrated waiting for a response—but how long that is depends both on the individual user and the situation. Actions which appear simple to the user, such as highlighting a button when it is clicked on, should happen fast enough that they appear instantaneous[1]; more complex actions can take a more noticeable time. Furthermore, users will tend to accept unusually bad performance in unusual situations: many users have reported success with using VNC over analogue modem links[ATT], even though the performance is appalling, purely because VNC permitted them to do something that they could not do by any other means.

User-based studies into the usability of GUIs tend to focus on an extremely narrow issue, such as whether button bars are better than menus[ETRS95] or speech is better that keyboard acceleration keys[PL91], for particular applications. Such studies can return meaningful and useful results. Trying to create a study to evaluate whether an entire window system is fast enough would be very difficult, and would almost certainly *not* provide any useful results. Users interact with applications built above the

[1] According to Card et al.[CMN83], the fastest human visual response time is 50 milliseconds; the average is 100 ms. Simple GUI updates should occur more rapidly than that.

Number of invocations	Pixels per invocation	In-process on A	Cross-process on A	In-process on B	Cross-process on B	Display on A Client on B	Display on B Client on A
10000	1	6.87	15.87	1.63	3.01	11.96	8.45
1000	10	75.50	78.65	19.86	20.59	78.71	20.33
100	100	11.21	11.47	2.92	3.00	11.23	3.04
10	1000	1.19	1.30	0.30	0.34	1.11	0.60
1	10000	0.33	0.41	0.06	0.07	0.22	0.35

Table 7.1: Seconds taken to render 10000 pixels in an XDisplay

window system, so any results would apply to the applications involved rather than to the window system itself.

Experience with simple applications running in DAWS, where all parts of the system are within the same island of connectivity, shows that the user interface performance is very good. All of the Components which have been developed update their imagery sufficiently rapidly that changes appear instantaneous, even when running on relatively slow machines. There is no visible difference between the responsiveness of a GUI running within the XDisplay, and a GUI running natively within X. Whether this translates to more complex applications performing well depends greatly upon the applications themselves, but DAWS certainly provides the tools with which to build responsive applications. Performance when not all objects are within the same island is discussed in section 7.1.3.

7.1.1 Rendering performance

One aspect of performance for which it *is* possible to provide hard figures is that of rendering speed. Table 7.1 shows the times, in seconds, taken to render 10000 pixels in pseudo-random positions, with different colours, directly into an XDisplay virtual display with a resolution of 640 by 480 pixels. The times are presented for performing 10000 calls to `Point()`, 1000 calls to `CPoint()` with a sequence of 10 points, 100 calls to `CPoint()` with a sequence of 100 points, 10 calls to `CPoint()` with 1000 points, and finally a single call to `CPoint()` with a sequence of all 10000 points. Two different machines were tested: machine A was a 90 MHz. Pentium; machine B was a 450 MHz. Pentium II. Both machines were running the Linux operating system, had no other significant processing load, and were linked with an un-loaded 10 Mbit/s Ethernet. The figures show the results of running the test with the client and XDisplay in the same process, with them in separate processes, and across the network. Both machines used

Number of invocations	Pixels per invocation	In-process on A	Cross-process on A	In-process on B	Cross-process on B	Display on A Client on B	Display on B Client on A
10000	1	0.28	9.79	0.03	2.01	7.32	7.36
1000	10	0.25	1.30	0.03	0.25	0.95	1.03
100	100	0.23	0.39	0.03	0.06	0.25	0.37
10	1000	0.23	0.31	0.03	0.09	0.20	0.33
1	10000	0.24	0.29	0.03	0.04	0.14	0.26

Table 7.2: Seconds to render 10000 pixels to an invisible frame buffer

a 16 bits-per-pixel display, so the XDisplay implementation had to reduce the number of significant bits in every colour specified.

At first glance, the figures look rather odd, in that there is a huge peak in the time taken to perform 1000 operation invocations. The expected pattern would be for tests involving fewer invocations to take less time. The discrepancy is explained by the way the XDisplay performs its rendering: at the end of each operation which has modified the virtual frame buffer, X is instructed to repaint the bounding rectangle of the parts which have changed. The calls which render an individual point only require that a tiny portion of the X screen is updated for each operation invocation, so the overhead is quite low. When `CPoint()` is called with a sequence of ten pseudo-randomly placed points, the bounding rectangle will generally be quite large. This means that for each of the 1000 invocations, large parts of the X display are being repainted.

To get some idea of how a direct hardware-based display would perform, the same tests were repeated with rendering to X disabled; the rendering operations still reduced the colour-space to 16 bits-per-pixel, and stored the values into the virtual frame buffer. The results are shown in table 7.2. This time, the values take the expected form. Calling operations in-process on machine B is basically too quick for the test—the figure of 0.03 seconds is an artifact of the timing mechanism. A much larger number of points would have to be plotted to acquire meaningful data in that situation. The time increase with 10 invocations on cross-process calls with machine B was consistently measured over many test runs; it is probably due to some buffer size issue within the protocol stack. It is interesting to note that in the inter-machine tests, having the display on the slower machine consistently performs better than the opposite situation. This implies that generating the pseudo-random numbers is more time consuming than the drawing operations.

In summary, the performance of the display is quite acceptable for the vast majority of applications. As expected, it is in general better for clients to batch as much data as possible, so as to limit the effect of communications latency. The exception to this is that the screen update method used by the current XDisplay implementation can be very slow for updates of sparsely-spaced rendering operations. The XDisplay could attempt to be more intelligent in its updates, but the logic to do so would probably consume more time than would be saved.

Unfortunately, the system is not quite capable of performing updates at a rate suitable for video. An experimental program sending `RectMap` blocks of 320 by 240 pixels achieved a frame rate of about 30 frames-per-second on the 450 MHz. machine, with full rendering to the X screen, with the client and display in separate processes. Rendering just to the virtual frame buffer, and not to the X screen, results in a performance of around 90 frames-per-second. These results would be acceptable for small video windows, but such a powerful machine should be capable of much more. To play video in DAWS, it would be best to either send compressed data to a Resource on the display server, or even to use a non-CORBA stream protocol, negotiated through a CORBA interface.

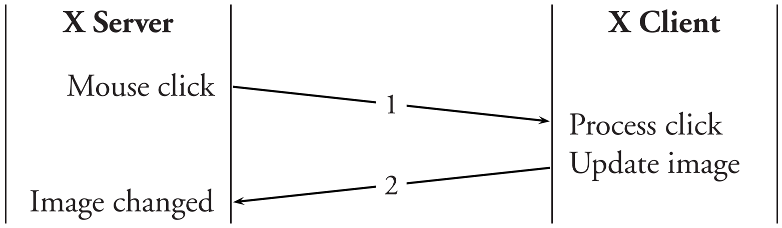
7.1.2 Synchronous invocation

As was explained in section 3.3, one of the aims of the DAWS design was to show that it is possible to build a responsive window system using a synchronous operation invocation system such as CORBA. The system which has been implemented has confirmed this point. In fact, it is not altogether surprising when the way in which X works is considered. Although X permits many protocol requests to be in transit at once, most user interface interactions actually involve a number of synchronous invocations.

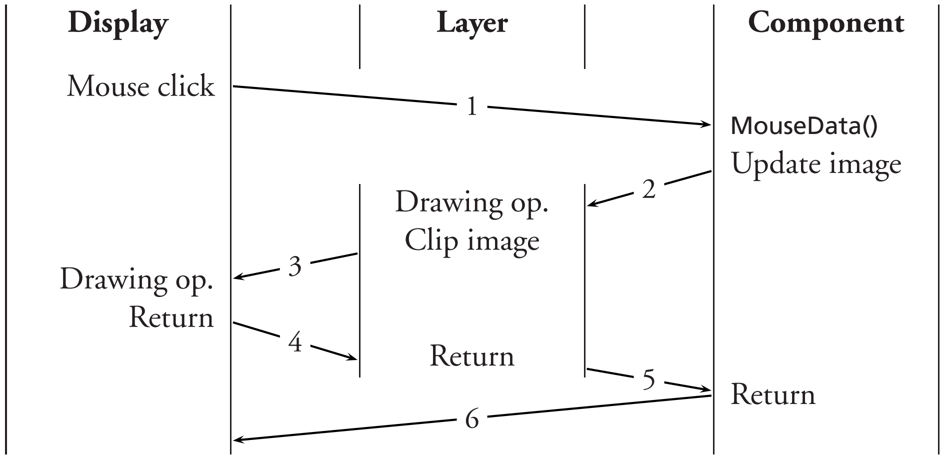
For example, compare the activity when the user clicks on a button within X and within DAWS. Figure 7.1 shows the communication involved in the two systems. In X, the server passes a mouse event to the widget implementation within the application, then the widget implementation sends X protocol requests to change the button's imagery. The client has no way to tell when the imagery has definitely been updated. The latency in updating the imagery is equivalent to two network hops.

In DAWS, a click on a button Component results in the display's mouse server invoking the Component's `MouseData()` operation. Within that operation implementation, the button Component performs a call to one of its Layer's drawing operations in order to update its imagery[2]. The Layer implementation clips the drawing operation and passes it on to the display's drawing Resource. At this stage, the button's on-screen imagery has been updated, but none of the CORBA operations have yet returned to their callers. The number of network transmissions these steps involve depends on where the various objects are placed. With all of the objects in different address spaces, it takes three network hops to update the button's imagery, and another

[2]It may, of course, require several drawing operation invocations to update the button's imagery, but that sort of situation is rare as long as high-level drawing Resources are provided.



a. X window system



b. DAWS

Figure 7.1: Network traffic when the user clicks on a button

three before all of the CORBA invocations have completed. Making the situation more similar to X's architecture by placing the Layer implementation in the same address space as the display reduces this to two hops, the same as X. Co-locating the Component implementation with the display removes *all* of the network hops.

The fact that the callers of DAWS operations usually wait for the result of the invocation before they make another call means that there is a limit on how soon another call is made. This can, at most, double the latency of the system.

One approach to reducing DAWS' latency would be to use CORBA's *deferred synchronous* operation invocation. This permits a client to perform an invocation and collect the result at some later time, meaning that it can perform a second invocation before the first has returned. The big difficulty with this approach is that DII, the dynamic invocation interface, must be used. This makes the code difficult to write, and sacrifices compile-time type safety. None of the parts of DAWS which have been implemented use deferred synchronous invocation, but it may be a useful technique in special circumstances which are highly latency dependent.

Latency (ms)	Object placement
< 20	Objects may be placed anywhere.
20–50	Layers and low-level Components (buttons, scroll-bars, etc.) should be migrated. Components performing layout operations can be remote.
50–100	Layout operations are clearly visible. Layout Components should be migrated for prolonged use.
>100	The LayerMgr should be migrated if it is under heavy use.

The above figures provide rules-of-thumb for object placement if there is a single significant network split in the arrangement of DAWS objects. This is the case if the user is in one island of connectivity while their applications are running in another. The way in which objects are placed within an island is largely irrelevant.

Table 7.3: Summary of DAWS latency dependence

7.1.3 Simulating network delay

In the local area, all of this discussion of latency is somewhat redundant since the delays never become large enough for there to be a difficulty. The interesting situations are therefore those in which not all objects involved in a session are within the same island of connectivity. There are significant practical difficulties in testing the system in a genuine wide area network, so network latency has been simulated by introducing delays into operation invocations.

It is quite simple to simulate latency in all operations of an object by adding a small amount of code to the object skeleton's `dispatch()` function (which was described in section 5.4.1). The delay code simply causes the thread performing the operation dispatch to sleep for a pseudo-random time within a specified interval. A more accurate simulation of network delay could be implemented by taking the message size into account, but such accuracy is not necessary. Due to the way in which threads are scheduled, there is a minimum time for which a thread can sleep. On Linux, this minimum is around 20ms; specifying a delay shorter than that results in a longer delay than requested.

The results of latency simulation are not at all surprising; they are summarised in table 7.3. Placing core objects such as Layers and low-level GUI Components (buttons, scroll bars, etc.) across links with high latency makes the system essentially unusable. The effects are apparent, but just about tolerable, when the latency is set to the minimum 20ms. Any more delay and the system becomes too unresponsive.

Other objects are more tolerant of latency. Higher-level Components which perform layout of other Components are only required when Layers change geometry, or parts of the display are repainted (since paint requests propagate through the Component hierarchy), so it is not completely unreasonable to place such Components across a slow connection. With delays of around 50ms, the order in which Components are painted is clearly visible; delays of 100ms start to become very obtrusive. Although the user would probably tolerate such a situation for brief use, it is difficult to imagine a situation in which the low-level Components could be migrated, but slightly higher-level Components could not.

More reasonable is the situation in which the LayerMgr is remote from the more latency-dependent objects. The LayerMgr is only involved when Layers move relative to each other, so once the user's desktop is arranged, the LayerMgr is required quite rarely. Re-ordering Layers only involves small numbers of messages to pass to and from the LayerMgr, so latencies of several hundred milliseconds are not unreasonable. Dragging a Layer to a new position requires a continuous stream of updates from the LayerMgr, but even in that case, the performance degrades quite acceptably as the latency increases—the movement of the dragged Layer becomes less smooth, but even with delays of several hundred milliseconds, it is not impossible to use. A useful migration strategy for the LayerMgr would be to migrate it only if it was receiving a large number of operation invocations.

The sensitivity to latency of the connection between the Components and the application code is of course highly dependent on the type of application. By creating Components which hold a cache of application state, as recommended in section 7.2 below, most applications can be made very insensitive to latency.

Measured network delay

To provide some context for the simulations, table 7.4 shows the ping times (found with Unix's 'ping' command) from a machine in Cambridge to locations around the world, using 1 kilobyte packets. The figures show the minimum, mean, and maximum times over ten pings. The results indicate that it would be possible for a user to roam anywhere within the Cambridge University network without really having to worry about migrating parts of their DAWS session. Roaming further, to other UK universities, or to other UK sites with good connectivity, would require that low-level objects were migrated, but higher-level objects could remain in the home location.

Moving further afield, parts of Europe, such as universities in Paris and Berlin, are quite close in network terms, and it would just about be possible to work after migrating only a few DAWS objects. Other places in Europe, such as Poland, have much greater latency.

At the time of writing, all traffic from the UK academic network (JANET) to the United States and beyond leaves the UK after the router named tglobe-gw2.ja.net, and emerges from the other side of the Atlantic at ppt-gw.ja.net, which is at Pennant Point

	Time in milliseconds		
Location	**Min.**	**Mean**	**Max.**
localhost	0.3	0.3	0.3
www (local web server)	2.3	2.5	2.5
www.cam.ac.uk	9.6	9.6	9.8
www.doc.ic.ac.uk	19.7	34.2	69.9
www.ed.ac.uk	25.2	36.2	64.4
www.demon.co.uk	22.0	29.1	45.6
www.ecp.fr	48.8	56.8	67.7
www.tu-berlin.de	51.0	56.1	76.2
www.krakow.pl	180.2	226.2	322.9
tglobe-gw2.ja.net	13.8	16.1	21.9
ppt-gw.ja.net	106.1	118.9	134.8
www.mit.edu	153.9	169.9	190.5
www.stanford.edu	204.1	211.6	232.4
www.usyd.edu.au	521.0	525.4	532.5

Table 7.4: Network ping times from Cambridge University to various locations

in Canada. The timing shows that the Atlantic accounts for around 50ms of delay in each direction. Coping with this much latency would basically require that almost all objects in a DAWS session were migrated. To complete the measurements, locations on the West coast of America have significantly more latency than those on the East coast. Finally, packets to Australia go all the way across the USA before going under the Pacific, resulting in a round-trip time of over half a second.

7.1.4 Multi-threading

One technique which is used throughout DAWS to reduce the impact of latency is to invoke operations on a number of objects concurrently using multiple threads. This approach is clearly beneficial when there is some genuine concurrency to be exploited, either due to the objects being distributed, or objects running on a multi-processor machine. It is also useful if there is some significant network delay between objects, since it permits several requests to be 'in-flight' across the network at the same time.

It is common for many objects to run on a single one-processor machine. In this case, there is some performance loss, compared to invoking operations on each object in sequence, due to the extra thread switching overhead. To evaluate the consequences of this situation, the HBox and VBox Components were implemented to be able to communicate with their sub-Components either with one thread per sub-Component, or with a single thread.

The two threading policies were tested on a number of different Component layouts. In the most complex test, the layout consisted of a VBox containing a total of 25 sub-Components, including three nested HBoxes. In this test, the sequential invocation approach set the geometry of all of the Components in an average of 111ms; the concurrent version took 185ms. 185ms would be noticed by the majority of users, but the way in which the Components are all drawn at roughly the same time makes the rendering *seem* rapid. The simpler layouts resulted in similar ratios between the times for the two methods.

To demonstrate one of the benefits of the concurrent approach, the test was repeated with a simulated latency of 20ms between the Components and the Layer into which they were rendering. In this case, the sequential version took 1859ms on average, whereas the concurrent version only took 705ms.

7.1.5 Summary

DAWS provides the facilities with which to build responsive user interfaces. If all of the objects involved in a session are within the same island of connectivity, in most circumstances it makes very little difference where within the island they are placed. The exceptions to this are, unsurprisingly, objects which require very low-latency communication, and high-bandwidth situations such as full-motion video playback.

When objects are distributed across a number of islands, object placement is much more important. Simulations have shown that the lowest-level objects must essentially always follow the user, but that there is more latitude with higher-level objects.

Finally, use of multiple threads can decrease the susceptibility of the system to latency, but doing so reduces performance when there is no distribution or latency.

7.2 Application design

This dissertation, and the work which it describes, has intentionally paid little attention to the design of applications which display their GUIs through DAWS. It would, indeed, be undesirable for a window system to dictate the architecture of the applications which make use of it. However, for the best possible performance when users are far from their home bases, there are a number of useful design patterns which can be followed.

Most applications can be thought of as having two sections: the core application logic, and the user interface logic. There are good software engineering arguments for making this split an explicit, rather than implicit one[Sze88]—in other words, there should be a clear distinction between parts of the application code which deal with core logic and parts which deal with the user interface, rather than having the two types intermingled. Applications which make this split can benefit most from DAWS since the user interface sections can more easily be migrated.

The simplest method for an application to make good use of DAWS is of course for it to use the standard DAWS Component system, rather than attempting to render directly into Layers. As explained in section 3.4.4, most low-level Components and simple layout Components will be supported at all locations, so applications which use them will automatically be able to migrate the lowest-level parts of their GUI functionality to the user's current location. In addition, the application should make use of events which are passed directly from Component to Component, rather than via the application, wherever possible. In this way, the number of cross-network invocations to the application's location is minimised, reducing the impact of latency on the application as a whole.

The use of low-level Components provides a significant improvement in latency dependence, but it still requires a large number of network round-trips for complex user interfaces. A powerful solution to this is for the application itself to create high-level Components which contain the majority of the user interface logic. So, for example, a spreadsheet application would create Components to manage the main view of the grid of cells, and Components for each of the application-specific dialogue boxes. By structuring the application in this way, the quantity of traffic from the Components to the application core is greatly reduced. These high-level Components should form a cache of the application's state, as explained in section 3.5.8, so failures at untrusted locations are not too damaging.

The problem with using application-specific high-level Components is that, unlike the low-level Components (and common high-level ones such as file choosers), it is quite unlikely that they will be available at all locations visited by the user. There are two main solutions to this. The first is to implement Components using a platform-independent technology such as Java, or a scripting language, so that code as well as state can migrate. The arguments against using Java for low-level parts of the system outlined in section 3.3.2 do not apply to higher-level parts, since they are unlikely to require any platform-specific knowledge, and raw processing speed is less important. The second solution is the far more ad-hoc approach of manually installing platform-dependent Components as and when they are required. When the user visited a new location, if the Components already supported at that location were not sufficiently high-level to provide good performance, application-specific Components could be manually installed on suitable machines.

Creating such high-level Components also has the benefit that it provides a clean method of splitting user interface functionality from core application functionality. If the Components' interfaces are sufficiently abstract, it would be relatively simple to replace the normal Component implementations with ones which provide text-only or speech interfaces, for example. This would be much harder if low-level Components were used directly.

Taking the DAWS philosophy even further, the application's core logic can be implemented using factories and Resources, so that significant parts, or even the entire application, can migrate to follow the user. Many applications will require at least some functionality to remain at the user's home base, to deal with facilities such as file storage. Another reason for leaving some parts of the application at the home base is that it is likely to be less prone to failures than a remote location, so it is best if all important application state is kept there.

7.3 Example configurations

Returning to the window system itself, this section describes two example configurations which illustrate the way in which DAWS' adaptive nature permits it to achieve the best possible performance when special-purpose display hardware is available.

7.3.1 3-d Rendering

The first example is hardware for rendering three-dimensional scenes. Many graphics cards now support 3-d operations themselves, and extension cards to add 3-d support to existing graphics devices are also available. Such hardware would be supported through one or more Resources of the `Display` object.

The `Display`'s Resource(s) could either provide an interface to just the operations supported by the hardware, or alternatively a general-purpose high-level set of 3-d operations, such as Silicon Graphics' OpenGL[SA98]. The former would be most suitable for 'thin' displays with little processing power; the latter would be best for high-powered general-purpose workstations. Clients of the 3-d rendering capabilities would prefer to be able to use the high-level style of interface, so self-contained versions of the high-level Resource would be made available through a ResourceFactory. These self-contained Resources would be able to adapt to a range of hardware-based Resources, and also to displays with no hardware support for 3-d at all.

Providing 3-d rendering Resources at the display is not sufficient for a working system, since all rendering must pass through a `Layer` object in order to be clipped to the Layer's current shape. The `Layer` object would therefore also have to support the 3-d Resource[3]. The Layer's Resource could perform all of the clipping, just as the current simple drawing Resources do, but that may be considerably less efficient than performing at least some of the clipping on the display's Resource. Some hardware may, in fact, have limited support for clipping the 3-d objects it is rendering, and all OpenGL implementations must support basic clipping. A more likely approach is therefore for the Layer's Resource to partially clip the incoming requests, by removing all 3-d objects which are completely outside the layer's viewable area for example, and to rely on either

[3] Alternatively, a remote ResourceFactory could be used to extend the Layers.

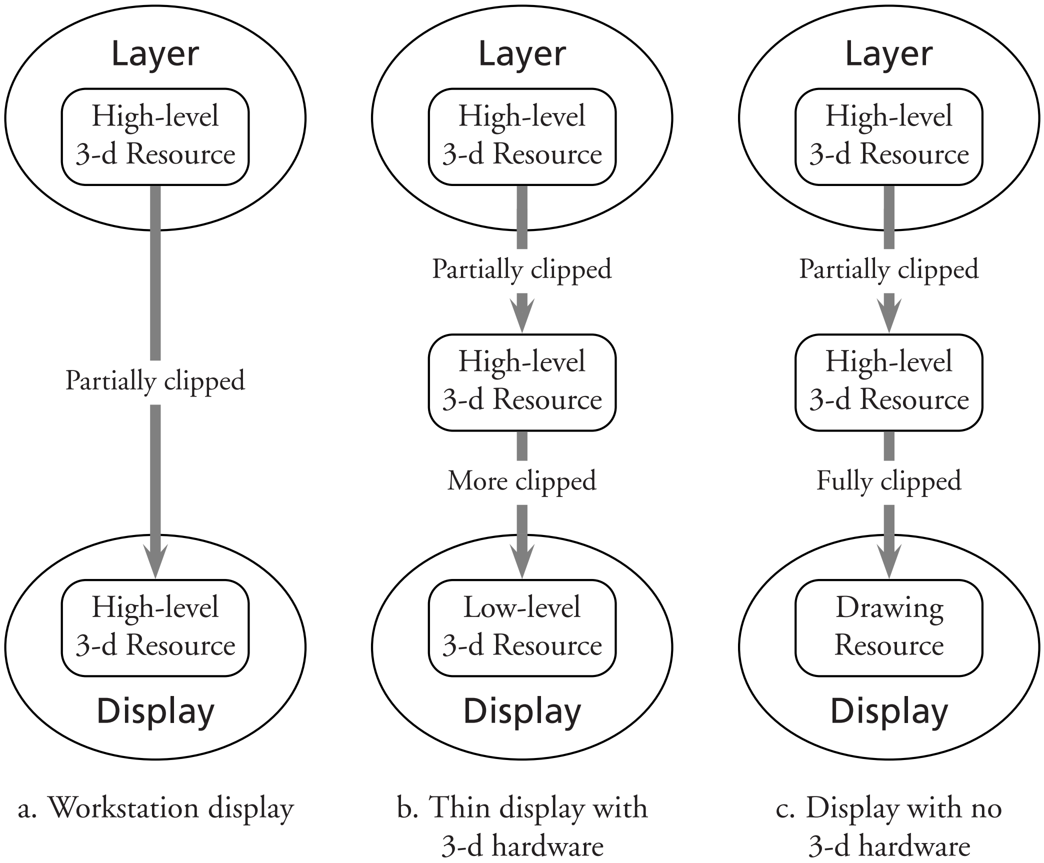


Figure 7.2: Example DAWS configurations for 3-d rendering

the display's high-level Resource (which may be remote from the display), or the hardware itself to perform the remaining clipping. Figure 7.2 shows three example configurations in which 3-d rendering is being performed. Only the Resources involved in 3-d rendering are shown for clarity.

7.3.2 General-purpose clipping hardware

Some systems, such as the Desk-Area-Network frame store mentioned in section 2.4.4, have hardware capable of performing general-purpose clipping operations. One way that DAWS could make use of such hardware would be to create special Layer object implementations which ran within the same address space as the Display object, permitting them to interface with the clipping hardware. When a Display object migrated to a display with clipping hardware, all of the Layer objects would also be migrated to the same address space.

This approach would be inflexible since Layers are responsible for more than just clipping—input management in particular. Also, if the user has a complex desktop arrangement, there may be many more Layers in existence than are visible at any given time. Forcing the display machine to cope with large numbers of Layer objects supporting a wide range of functionality would require that displays with clipping hardware were quite powerful general-purpose machines. This would not necessarily be the case.

A far better solution would be to add a clipping Resource to the Display object, which would permit Layer objects, wherever they were running, to specify clipping regions within the display. When a Layer first connected to a display, and when it was notified that the display information had changed due to a migration, it would check to see whether the clipping Resource was available and, if so, make use of it. As with the clipping of 3-d shapes discussed above, it would probably be sensible for the Layer object to perform some simple clipping, such as clipping to the Layer's bounding rectangle, before passing operations on to the clipping Resource.

In the absence of clipping hardware, it would still be possible to provide a clipping Resource at the display, either directly or though a ResourceFactory, but this would probably not be worthwhile. Any display capable of supporting a software clipping Resource would almost certainly also be capable of supporting complete Layer objects, so the Resource would be redundant. Using a remote clipping Resource would cause greater latency due to the increased number of network hops. The best solution would be for the Layer objects to perform their own clipping operations, unless a hardware-based clipping Resource was available.

7.3.3 Summary

This section has shown how the DAWS Resource mechanism can be used to extend the core functionality of the window system, to cope with additional hardware facilities, and how parts of the system are able to adapt at run-time to make use of such extensions. The two example configurations, like most hardware-based enhancements, involve adding Resources to the main Display object.

One significant difference between the two examples is that all existing parts of the system would automatically make use of a hardware clipping device, whereas Components (or other elements of the system) would have to be written to make use of a new 3-d rendering Resource. This observation leads to a classification of two types of DAWS extension: those which provide new implementations of existing functions, and those which add brand new functions. When new hardware-driven functions are added, it is important to also create self-contained software versions of the Resources, created by ResourceFactories, so that new clients can work with existing displays.

7.4 Integration with existing window systems

The discussion so far has assumed that DAWS would be used in isolation, running natively on all platforms, from high-powered workstations to simple thin displays. This is, realistically, highly unlikely to ever be the case, since there is such a large installed base of systems running other window systems, mainly Microsoft Windows, and the X window system. It is not unreasonable, on the other hand, to expect to find DAWS running natively on new, thin, display types. It would therefore be useful to integrate DAWS with Windows and X, so that users may migrate from native DAWS displays to non-DAWS displays, and vice-versa.

One approach to such integration has, in fact, already been implemented in the form of the XDisplay virtual display. Virtual displays like this, and VNC, work by pretending that all displays are incredibly primitive, and only make use of the lowest-common-denominator functionality. Virtual displays will inevitably never perform as well as the display's native window system.

A far better approach would be to provide new implementations of much larger parts of DAWS, which were aware of the host window system. These implementations would, in effect, be wrappers around the host window system's facilities which made them appear to be DAWS facilities. This approach could be taken to a number of levels:

- Re-implement just the `LayerMgr` and `Layer` objects.

  This would permit DAWS windows to co-exist with other windows from the host window system. The host window system would perform clipping, but all Layer contents would be drawn using DAWS drawing operations.

- Re-implement low-level DAWS Components using the host system's own widgets.

  In this way, the Components would probably perform more efficiently, especially in the case of Windows, where such elements are tightly coupled with the OS. Using the host widget set would also improve the consistency of imagery and behaviour between DAWS applications and native applications.

- Re-implement high-level DAWS Components, such as file choosers using the host window system's facilities.

  This would do little to improve performance, but would further improve consistency between DAWS applications and native ones.

There are also a number of peripheral issues, such as whether the host window system's text rendering support should be used instead of DAWS' own system. By far the greatest benefit would be reaped from the use of the host window system's widgets.

An interesting and useful side-effect from this integration approach is that it would provide a highly flexible basis for creating cross-platform applications. All DAWS applications would be portable at a network level, in that an application running on Windows could export its display to any DAWS-capable display, and any DAWS application could display on Windows. It would not require too much effort to create applications which were also portable at the source code level. This scheme has the significant advantage over similar systems, such as Java's AWT[GM95], that unusual facilities of the underlying window system can be accessed by applications which know about them, through the Resource mechanism.

7.5 Summary

This chapter has evaluated the performance of DAWS, and concluded that, in the local area, it performs as well as other window systems. Simulations of the latency that would be involved in using DAWS in a wide area have shown that, not surprisingly, low-level parts of the window system must always migrate to within the user's current island of connectivity. Acceptable performance is only achieved when the latency to all low-level Components is less than about 20ms.

By studying a number of example situations, this chapter has also demonstrated the extensible and adaptive nature of DAWS. The Resource mechanism permits DAWS to be extended to support new features such as 3-d rendering, either using software or hardware. The use of Resources also allows existing low-level functions to be enhanced with special-purpose, unusual, hardware.

As mentioned in Chapter 1, there are benefits to allowing applications to migrate to follow the user. The DAWS framework of factories and Resources can be used by applications which wish to do this.

Finally, a scheme by which DAWS could be integrated with existing window systems, such as Microsoft Windows and X, has been described. This scheme means that, not only can DAWS applications export their interfaces to other window systems in an efficient manner, but also that applications become portable between different window systems. Unlike other application portability systems, the portability of DAWS-based interfaces does not come at the expense of making all target window systems appear identical—the Resource mechanism means that applications can make use of special features which some systems have but others do not.

8 Conclusions and Further Work

This dissertation has presented a new CORBA-based window system which supports users who are mobile in a wide area. This chapter summarises the work and its conclusions, and makes suggestions for further areas of study.

8.1 Summary

DAWS grew out of the desire to support users who are mobile in a very wide area, by permitting them to access their normal desktop environment from any networked computer. As explained in Chapter 2, existing network-aware window systems cannot be extended to long-distance access since they are too dependent on latency. This is because the protocols of these systems split the functionality of the window system so that most, if not all, of the interactive behaviour of the GUI is performed by libraries within the applications, rather than by the end-point display.

DAWS demonstrates that, by using a component-based framework of mobile objects, it is possible to build a responsive window system which is able to adapt to a far wider range of operating parameters than existing systems. Existing network-aware window systems use rigidly-defined protocols for the communication between applications and the window system, so they are not able to adapt to environments that their designers did not anticipate. The use of CORBA means that the 'protocol' used by DAWS can, in effect, be dynamically extended and adapted.

DAWS has a radical new architecture in which a network split may be made between any of the functional elements of the window system. Each functional element is a CORBA object which is able to migrate from location to location using the system detailed in Chapter 5. As explained in Chapter 3, objects are responsible for transferring their own state during migrations. Additionally, the migration system does not make use of mobile code, meaning that objects may migrate to different implementations of the same functionality.

Experience in the local area has shown that, for most parts of the system, it does not matter where within an island of connectivity objects are placed—the system always has acceptable performance. Simulations of network latency have shown that, not surprisingly, when the user moves out of their home island of connectivity, all of the GUI Components responsible for interactive behaviour must migrate to follow them.

The fine-grain distribution of low-level parts of DAWS is therefore not useful for reducing the window system's dependence of latency—a monolithic window system server which supported interactive Components would be sufficient. However, the use of low-level distribution *is* extremely useful for supporting the wide range of devices that the user may wish to use, from 'thin' network-attached displays capable of running only the lowest-level objects, through to high-powered general-purpose workstations capable of running all parts of the window system.

DAWS achieves its adaptability and extensibility through the ResourceFinder and ResourceFactory mechanisms. Resources provide a general way for a CORBA interface to be dynamically extended without the complications associated with multiple inheritance. Use of Resources permits DAWS to supply interfaces to optional hardware facilities, such as 3-d rendering, and to transparently support esoteric hardware capabilities such as image clipping.

The Resource mechanism is generally applicable to a wide range of different distributed systems problems, and is therefore one of the most significant contributions of this work.

8.2 Future work

Prototypes of all of the low-level parts of DAWS have been created, but many of the higher-level parts, especially a complete set of GUI components, are yet to be written. As well as completing these parts so that DAWS can be used as a full window system, there are a number of other interesting areas for research with the window system:

- DAWS is, at present, completely written in C++, so migrating unusual objects can be difficult from a management point-of-view. Using Java or scripting languages for high-level Components within DAWS would remove this difficulty. Scripting languages would also probably result in much more rapid Component development.

- This work has purposefully avoided the issue of how the system decides when to migrate objects, and where best to place them. There is great potential for research in this area.

- The DAWS design assumes that the user always has access to some form of fixed computing infrastructure. It would be interesting to see if DAWS could be extended to support disconnected operation for users with notebooks and PDAs.

This could probably be achieved by exploiting the way in which Components implement a cache of application state.

- As mentioned in Chapter 7, DAWS could be integrated with existing window systems. The implications of this on application portability and flexibility may be worth studying.

- It would be interesting to evaluate whether there is any significant benefit to constructing DAWS applications in such a way that large parts of the application functionality can migrate.

Finally, there is great scope for future research using the DAWS framework, especially the Resource mechanism, to create highly extensible and adaptable distributed systems for a wide range of problems.

A
CORBA Life Cycle Service IDL

```
#include <Naming.idl>

module CosLifeCycle {
  typedef CosNaming::Name     Key;
  typedef Object              Factory;
  typedef sequence <Factory> Factories;

  typedef struct NVP {
    CosNaming::Istring  name;
    any                 value;
  } NameValuePair;
  typedef sequence <NameValuePair> Criteria;

  exception NoFactory          { Key       search_key;        };
  exception NotCopyable        { string    reason;            };
  exception NotMovable         { string    reason;            };
  exception NotRemovable       { string    reason;            };
  exception InvalidCriteria    { Criteria  invalid_criteria;  };
  exception CannotMeetCriteria { Criteria  unmet_criteria;    };

  interface FactoryFinder {
    Factories find_factories(in Key factory_key)
      raises(NoFactory);
  };

  interface LifeCycleObject {

    LifeCycleObject copy(in FactoryFinder there,
                         in Criteria      the_criteria)
      raises(NoFactory, NotCopyable,
             InvalidCriteria, CannotMeetCriteria);

    void move(in FactoryFinder there,
              in Criteria      the_criteria)
```

```
      raises (NoFactory, NotMovable,
              InvalidCriteria, CannotMeetCriteria);

    void remove()
      raises (NotRemovable);
  };

  interface GenericFactory {
    boolean supports(in Key k);

    Object create_object(in Key k, in Criteria the_criteria)
      raises (NoFactory, InvalidCriteria, CannotMeetCriteria);
  };
};
```

B
TextRender Details

The following shows a DAWS font configuration file. Each line contains the DAWS name of the font followed by the full path name of the TrueType file, possibly followed by a collection index, and finally some number of flag letters indicating the font's properties. TrueType collections are font files which contain more than font design. The flags are 'B', 'I', 'S', 'F', and 'R', to indicate bold, italic, small-caps, fixed-width, and right-to-left.

```
ArialBlack          $(DAWS)/fonts/ariblk.ttf          B
ComicSans           $(DAWS)/fonts/comic.ttf
ComicSansBold       $(DAWS)/fonts/comicbd.ttf         B
Georgia             $(DAWS)/fonts/georgia.ttf
GeorgiaBold         $(DAWS)/fonts/georgiab.ttf        B
GeorgiaItalic       $(DAWS)/fonts/georgiai.ttf        I
GeorgiaBoldItalic   $(DAWS)/fonts/georgiaz.ttf        BI
Impact              $(DAWS)/fonts/impact.ttf
Monotype.com        $(DAWS)/fonts/monotype.ttf        F
Trebuchet           $(DAWS)/fonts/trebuc.ttf
TrebuchetBold       $(DAWS)/fonts/trebucbd.ttf        B
TrebuchetBoldItalic $(DAWS)/fonts/trebucbi.ttf        BI
TrebuchetItalic     $(DAWS)/fonts/trebucit.ttf        I
Verdana             $(DAWS)/fonts/verdana.ttf
VerdanaBold         $(DAWS)/fonts/verdanab.ttf        B
VerdanaItalic       $(DAWS)/fonts/verdanai.ttf        I
VerdanaBoldItalic   $(DAWS)/fonts/verdanaz.ttf        BI
MingLiu             /home/jnw/DAWS/fonts/mingliu.ttf
```

Most of the fonts described in the configuration file are taken from Microsoft's free web font distribution available at http://www.microsoft.com/typography/fontpack/. Figure B.1 shows some of them rendered into an XDisplay virtual display.

Figure B.1: Some example rendered text

C
Quill Grammar

The following is the full grammar for the Quill GUI description language described in section 6.5.

```
<expression>        ::= <component_spec>
                    |   <component_spec> “{” <expression_list> “}”
<expression_list>   ::= <expression>
                    |   <expression> <expression_list>
<component_spec>    ::= <component_type> [<instance_spec>] [<argument_spec>]
<component_type>    ::= <identifier>
<instance_spec>     ::= “<” <instance_name> “>”
<instance_name>     ::= <identifier>
<argument_spec>     ::= “(” <argument_list> “)”
<argument_list>     ::= <argument>
                    |   <argument> “,” <argument_list>
<argument>          ::= <argument_name> “=” <argument_value>
<argument_name>     ::= <identifier>
<argument_value>    ::= <long_value>
                    |   <string_value>
                    |   <objref_value>
                    |   <colour_value>
                    |   <eventhandler_value>
                    |   <size_value>
                    |   <glue_value>
<long_value>        ::= <integer>
<string_value>      ::= “"” <string> “"”
```

<objref_value> ::= <identifier>

<colour_value> ::= “col(” <integer> “,” <integer> “,” <integer> “)”
| “col(” <hexadecimal> “)”

<eventhandler_value> ::= “ehandler(” <objref_value> “,” <eventhandler_key> “)”

<eventhandler_key> ::= *<string>*

<size_value> ::= “size(” <size_spec> “,” <size_spec> “)”

<glue_value> ::= “glue(” <glue_list> “)”

<glue_list> ::= <size_spec>
| <size_spec> “,” <glue_list>

<size_spec> ::= <integer>
| <integer> <integer> <integer>

<integer> ::= [“-”] [0–9]+
| <hexadecimal>

<hexadecimal> ::= “0x” [0-9a-f]$^{+}$

<identifier> ::= [a-zA-z][a-zA-Z0-9_]*

Bibliography

[Ado90] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, 2nd edition, 1990.

[Ado93] Adobe Systems Incorporated. *Programming the Display PostScript System with X*. Addison-Wesley, 1993.

[App97] Apple Computer Inc., 1 Infinite Loop, Cupertino, CA. *TrueType Reference Manual*, 1997. Available from http://fonts.apple.com/TTRefMan/.

[ASH97] Noha Adly, Pete Steggles, and Andy Harter. SPIRIT: A resource database for mobile users. Technical Report 97.2, Olivetti and Oracle Research Laboratory, 1997.

[ATT] Comments on VNC list. http://www.uk.research.att.com/vnc/archives/.

[ATT96] The Videotile. Web page: http://www.uk.research.att.com/tile.html, 1996.

[Bac97] Jean Bacon. *Concurrent Systems: operating systems, database and distributed systems: an integrated approach*. Addison-Wesley, second edition, 1997.

[Bar96] Paul R. Barham. *Devices in a Multi-Service Operating System*. PhD thesis, University of Cambridge, 1996.

[Bat98] John Bates. The state of the art in distributed and dependable computing. Technical report, Laboratory for Communications Engineering, University of Cambridge, October 1998.

[BC96] Krishna A. Bharat and Luca Cardelli. Migratory applications. Technical Report 138, Digital Systems Research Center, February 1996.

[Ber98] Berlin Consortium home page. http://www.berlin-consortium.org/, 1998.

[BHB96] John Bates, David Halls, and Jean Bacon. A framework to support mobile users of multimedia applications. *ACM Mobile Networks and Nomadic Applications*, 1996.

[BHR+98] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Straßer. Mole 3.0: A middleware for Java-based mobile software agents. In *Middleware 98*, 1998.

[BJW87] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. Technical Report 24, Digital Systems Research Center, August 1987.

[Bla95] Richard Black. DAN Framestore: Technical. In *ATM Document Collection 4 (The Green Book)*, chapter 2. University of Cambridge Computer Laboratory, September 1995.

[BM94] Marc H. Brown and James R. Meehan. *The FormsVBT Reference Manual, Version 3.3.* Digital Equipment Corporation, June 1994.

[BT98] Adam Buble and Petr Tuma. Throughput test. Web page: http://www.kav.cas.cz/~buble/corba/comp/test/through/, 1998.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Systems Research Center, 1994.

[CHY+97] P. Emerald Chung, Yennun Huang, Shalini Yajnik, et al. DCOM and CORBA side by side, step by step, and layer by layer. Technical report, Bell Laboratories, September 1997.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Cor92] David Cornelius. Xremote: a serial line protocol for X. In *Proceedings of the 6th Annual X Technical Conference*, January 1992.

[CPHW97] Steve B. Cousins, Andreas Paepcke, Scott W. Hassan, and Terry Winograd. Towards wide-area distributed interfaces. http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0037, 1997.

[Cra86] Stephen C. Crawley. *The Entity System: An Object-based Filing System.* PhD thesis, Computer Laboratory, University of Cambridge, 1986.

[CTD98] CTD Resource Network, Inc. Alternative keyboard FAQ—chording keyboards. Typing Injury FAQ, http://www.tifaq.org/keyboards/chording-keyboards.html, November 1998.

[Cus93] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993.

[Dan94] John Danskin. Higher bandwidth X. In *Proceedings of ACM Multimedia '94*, pages 89–96, San Francisco, California, October 1994.

[Dan95] John Danskin. Previewing PostScript over a telephone in 3 seconds per page. In *Proceedings of the 9th Annual X Technical Conference*, pages 23–40, January 1995.

[Dat97] DataBeam Corporation. A primer on the T.120 series standard. White Paper, May 1997.

[Dea98] Alan Dearle. Toward ubiquitous environments for mobile users. *IEEE Internet Computing*, 2(1):22–32, January–February 1998.

[DR88] N. Donato and R. Rocchetti. Techniques for manipulating arbitrary regions. *Course Notes 11 for SIGGRAPH 88, Atlanta, GA*, August 1988.

[EEB67] William K. English, Douglas C. Engelbart, and Melvyn L. Berman. Display-selection techniques for text manipulation. In *IEEE Transactions on Human Factors in Electronics*, volume HFE-8, number 1. IEEE, March 1967.

[ETRS95] Jason Ellis, Chi Tran, Jake Ryoo, and Ben Shneiderman. Buttons vs. menus: An exploratory study of pull-down menu selection as compared to button bars. Technical Report CS-TR-3452, Human-Computer Interaction Laboratory, Department of Computer Science, University of Maryland, June 1995.

[FK93] Jim Fulton and Christopher Kent Kantarjiev. A report on Low Bandwidth X (LBX). In *Proceedings of the 7th Annual X Technical Conference*, page 251, January 1993.

[Fri87] Peter Fritzson. Window system architectures — an overview. Technical report, Institutionen för datavetenskap, Universitet i Linköping och Tekniska Högskolan, 581 83 Linköping, Sweden, September 1987.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[Get98] Jim Gettys. The two edged sword. Editorial, Freshmeat Web Magazine, http://freshmeat.net/, October 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GM90] Hania Gajewska and Mark S. Manasse. Why X is not our ideal window system. *Software Practice and Experience*, 20(S2), October 1990.

[GM95] James Gosling and Henry McGilton. The Java language environment. Technical report, Sun Microsystems Computer Company, October 1995.

[Gos86] James Gosling. SunDEW: A Distributed and Extensible Window System. In *Proceedings of the 1986 Winter Usenix Technical Conference*, Denver, Colorado, 1986. Reprinted in [Fri87].

[GRA89] James Gosling, David S.H. Rosenthal, and Michelle J. Arden. *The NeWS Book: an Introduction to the Network/extensible Window System*. Sun technical reference library. Springer Verlag, 1989.

[Har98] Ben Harris. Interoperability between omniORB2 and Java ORBs. Web page: http://www.uk.research.att.com/omniORB/javaORBs.html, March 1998.

[Hen98] Michi Henning. Binding, migration, and scalability in CORBA. *Communications of the ACM*, October 1998.

[HL82] Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4, No. 4:527–551, October 1982.

[Hol90] David A. Holzgang. *Display PostScript Programming*. Addison-Wesley, 1990.

[Hop94] Don Hopkins. *The UNIX-HATERS Handbook*, chapter 7, The X-Windows Disaster. IDG Books, Programmers Press, June 1994.

[HR93] Graham Hamilton and Sanjay Radia. Using interface inheritance to address problems in system software evolution. Technical Report TR–93–21, Sun Microsystems Laboratories, Inc., November 1993.

[HvST97] Franz J. Hauck, Maarten van Steen, and Andrew S. Tanenbaum. A location service for worldwide distributed objects. Technical report, Dept. of Math. and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1997.

[Jac92] Christian P. Jacobi. Migrating widgets. In *Proceedings of the 6th Annual X Technical Conference*, January 1992.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6, No. 1, February 1988.

[KDF+93] Christopher Kent Kantarjiev, Alan Demers, Ron Frederick, Robert T. Krivacic, and Mark Weiser. Experiences with X in a wireless environment. Technical report, Xerox Palo Alto Research Center, September 1993.

[Kir98] Michael Kircher. Personal correspondence, July 1998.

[KP98] Michael Kircher and Irfan Pyarali. Location forwarding. Web page: http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/forwarding.html, 1998.

[LCW+88] Brian Lewis, Dan Craft, Bob Weissman, Veronica Falcao, and Randy Strauss. An evaluation of window system toolkits. Technical report, Olivetti Software Technology Laboratory, 1988.

[LMB+97] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *Journal of Selected Areas of Communications*, June 1997.

[LN84] K. A. Lantz and W. I. Nowicki. Structured graphics for distributed systems. In *ACM Transactions on Graphics*, volume 3, number 1, pages 23–51. ACM, January 1984.

[Lo98] Sai-Lai Lo. *The omniORB2 User's Guide*. AT&T Laboratories Cambridge, 1998. See http://www.uk.research.att.com/omniORB/.

[LP98] Sai-Lai Lo and Steve Pope. The implementation of a high performance ORB over multiple network transports. In *Proceedings of Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, September 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996.

[MB76] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(5):395–404, July 1976.

[MET98] META Group Consulting. CORBA vs. DCOM: Solutions for the enterprise. White Paper, March 1998.

[Mic96] Microsoft. DCOM technical overview. White Paper, 1996.

[Mic98a] Microsoft. DCOM architecture. White Paper, 1998.

[Mic98b] Microsoft. Microsoft Windows NT Server, Terminal Server Edition, version 4.0: An architectural overview. White Paper, June 1998.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST '95: Proceedings of the Eighth ACM Annual Symposium on User Interface Software and Technology*, pages 111–120, Pittsburgh, Pennsylvania, November 1995.

[Now85] W. Nowicki. *Partitioning of Function in a Distributed Graphics System*. PhD thesis, Department of Computer Science, Stanford University, California, 1985.

[Obj97] ObjectSpace Inc. ObjectSpace Voyager core package technical overview. White Paper, December 1997.

[OMG] Object Management Group home page. http://www.omg.org/.

[OMG96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1996.

[OMG97] Object Management Group. *Meta Object Facility (MOF) Specification*, 1997. OMG Document ad/97-10-02.

[OMG98a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, 1998.

[OMG98b] Object Management Group. *CORBAServices: Common Object Services Specification*, 2.2 edition, 1998.

[Pac94] Keith Packard. Designing LBX. In *Proceedings of the 8th Annual X Technical Conference*, pages 121–133, January 1994.

[Per96] C. Perkins, editor. *IP Mobility Support*. RFC 2002, October 1996.

[Pet97] Yvan Peter. Mobility management in CORBA: A generic implementation of the LifeCycle service. In *XXIV-th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'97)*, Milovy, Czech Republic, November 1997.

[Pik83] Rob Pike. Graphics in overlapping bitmap layers. *ACM Transactions on Graphics*, 2, No. 2:135–160, April 1983.

[Pik91] Rob Pike. $8\frac{1}{2}$, the Plan 9 window system. In *Proceedings of the Summer 1991 USENIX Conference*, pages 257–265, Nashville, 1991.

[PL91] Randy Pausch and James H. Leatherby. Voice input vs. keyboard accelerators: A user study. Technical Report TR-91-22, Computer Science Department, University of Virginia, October 1991.

[Pop96] Steven L. Pope. *Application Support for Mobile Computing*. PhD thesis, Computer Laboratory, University of Cambridge, October 1996.

[RBMH93] Tristan Richardson, Frazer Bennett, Glenford Mapp, and Andy Hopper. Teleporting in an X Window System environment. Technical report, Olivetti Research Laboratory, November 1993.

[RLML86] Jim Rees, Paul H. Levine, Nathaniel Mishkin, and Paul J. Leach. An extensible I/O system. In *Proceedings of the Summer 1986 Usenix Conference, Atlanta, Georgia*, 1986. Available from http://http://www.mit.edu/afs/athena.mit.edu/user/r/e/regis/links/macum%ich/Public/html/apollo/ost.ps.gz.

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, Computer Laboratory, University of Cambridge, April 1995.

[RSD+87] William Roberts, Mel Slater, Kieron Drake, Antony Simmins, Allan Davison, and Peter Williams. First impressions of NeWS. Technical report, Department of Computer Science, Queen Mary College, 190 Mile End Road, London E1 4NS, UK, August 1987.

[RSFWH98] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. Technical Report 98.1, Olivetti and Oracle Research Laboratory, 1998.

[SA88] Douglas K. Smith and Robert C. Alexander. *Fumbling the Future: How Xerox Invented, then Ignored, the First Personal Computer*. William Morrow and Company, Inc., New York, 1988.

[SA98] Mark Segal and Kurt Akely. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc., Mountain View, CA, 1.2.1 edition, October 1998.

[Sch98] Douglas C. Schmidt. Real-time CORBA with TAO (The ACE ORB). Web page: http://www.cs.wustl.edu/~schmidt/TAO.html, 1998.

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. In *ACM Transactions on Graphics*, volume 5, number 2, pages 79–109, April 1986.

[SG92] Robert W. Scheifler and Jim Gettys. *X Window System*. Digital Press, Bedford, Massachussetts, 1992.

[SH97] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. Technical report, Department of Computer Science, University of British Columbia, March 1997.

[SJ95] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, December 1995.

[Ste89] J. Steinhart. Introduction to window management. *Course Notes 11 for SIGGRAPH 89, Boston, MA*, August 1989.

[Sun97a] Sun Microsystems. Java Native Interface specification. Java Development Kit 1.1.7 Documentation, May 1997.

[Sun97b] Sun Microsystems. Java object serialization specification. Java Development Kit 1.1.7 Documentation, 1997.

[Sun97c] Sun Microsystems. *JavaBeans*, 1.01 edition, July 1997.

[Sun98] Sun Microsystems. Java Remote Method Invocation specification, July 1998.

[SWH98] Pete Steggles, Paul Webster, and Andy Harter. The implementation of a distributed framework to support 'follow me' applications. Technical Report 98.8, Olivetti and Oracle Research Laboratory, 1998.

[Sze88] Pedro Szekely. *Separating the User Interface from the Functionality of Application Programs*. PhD thesis, Carnegie Mellon University, 1988.

[Tur98] David Turner. *The FreeType Engine Core Library User Guide*, 1998. See http://www.freetype.org/.

[Uni96] Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996.

[W3C98] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, February 1998.

[WH99] James N. Weatherall and Andy Hopper. Predator: A distributed CORBA location service and example applications. Technical report, Laboratory for Communications Engineering, Engineering Department, University of Cambridge, 1999. Report in progress.

[WHFG92] Roy Want, Andy Hopper, Veronica Falcao, and Jonathon Gibbons. The Active Badge location system. Technical Report 92.1, Olivetti Research Ltd., 1992.

[XPr90] O'Reilly & Associates, Inc. *X Protocol Reference Manual*, second edition, May 1990.

Index