Y: A Successor to the X Window System

Mark Thomas
`<mbt99@doc.ic.ac.uk>`

Project Supervisor:
D. Rückert
`<dr@doc.ic.ac.uk>`

Second Marker:
E. Lupu
`<ecl1@doc.ic.ac.uk>`

June 18, 2003

Abstract

UNIX desktop environments are a mess. The proliferation of incompatible and inconsistent user interface toolkits is now the primary factor in the failure of enterprises to adopt UNIX as a desktop solution.

This report documents the creation of a comprehensive, elegant framework for a *complete* windowing system, including a standardised graphical user interface toolkit. 'Y' addresses many of the problems associated with current systems, whilst keeping and improving on their best features.

An initial implementation, which supports simple applications like a terminal emulator, a clock and a calculator, is provided.

Acknowledgements

Thanks to Daniel Rückert for supervising the project and for his help and advice regarding it.

Thanks to David McBride for his assistance with setting up my project machine and providing me with an ATI Radeon for it. Thanks to Philip Willoughby for his knowledge of the POSIX standard and help with the GNU Autotools and some of the more obscure libc functions. Thanks to Andrew Suffield for his help with the GNU Autotools and Arch.

Thanks to Nick Maynard and Karl O'Keeffe for discussions on window system and GUI design. Thanks to Tim Southerwood for discussions about possible features of Y. Thanks to Duncan White for discussions about the virtues of X.

Contents

Chapter 1

Introduction

The X Window System [23] is the *de facto* standard graphical user interface (GUI) system on UNIX and UNIX-like platforms such as GNU/Linux. However, as X approaches its 20th year, signs of its age are beginning to show. Commonly cited problems with X include:

- **X is too slow.** This is commonly dismissed as nonsense due to the high throughput that tweaked implementations of X have been proven to achieve[1]. What this does not take into account is that in the general case it is latency that matters more than throughput [6]. Unfortunately, the design of X does not facilitate low latency.

- **X places too much burden on the programmer.** The X protocol, and its corresponding library `Xlib`, provide very low level operations. As a result, programming directly with Xlib is very difficult. For this reason, programmers usually choose to use a toolkit library.

- **X has no standard toolkit.** In 1984, before GUIs were common-place, not providing a standard toolkit was the best way to achieve enough flexibility to create all the applications that had not yet been conceived. However, these days, with the benefit of the last two decades of experience [16, 25], it is much better to provide a complete set of standard user interface components that look and behave consistently.

  Aside from the user interface inconsistency, the lack of standard components also makes internationalisation difficult, particularly for languages which require a complex input method.

- **X is reaching the end of its life span.** XFree86, the most popular version of X that is in use, is now over 10 years old. Over the years it has been extended and modified many times, to the point where it is an incoherent mess.

  Although the X protocol supports extensions very well, some of the latest extensions have begun to interfere with each other. For example, when

[1]For example, 265,000 lines per second, each 100 pixels in length, determined using the `x11perf` program that comes with XFree86 [29], on an Intel Pentium 4 1.8 GHz CPU with an ATI Radeon VE graphics accelerator, using XFree86's `radeon` driver.

Xinerama (the extension which allows X desktops to span multiple monitors) was first released, it broke XVideo (the extension which allows X to use hardware accelerated overlays for video play back). The 'fix' for this was to allow XVideo to only work on the primary display. The latest extension, XRandR (Rotate and Resize), is also known to break many older applications which assume that the screen size will never change.

Further, the internal design of X itself is outdated. Even adding a simple feature, such as translucent windows, requires large changes to the server [17]. Because of the requirement to be backwardly compatible, these features must be implemented for everything that X works on, including two-colour displays.

- **X is too complex.** The years of extension and modification of the X protocol itself have left the unfortunate legacy that X is too complex. Additional protocols like ICCCM which have been layered above X in an attempt to solve problems have caused additional problems when it comes to understanding what is actually happening [24]. The xine media player for Linux has to probe which window manager is currently running and guess at the best way to switch to full screen. The developers gave up trying to find a consistent way to switch off the screen saver, and switched to the ugly hack of simulating the left shift key being pressed once every thirty seconds [7].

However, X has some useful features which any worthy replacement must also include:

- **Network Transparency.** X allows applications which are not running on the same machine to connect to the display (for a comparison between this and remote desktop access, see appendix B). This is a useful feature, particularly for administrators of servers that do not have displays of their own.

- **Modularity and Extensibility.** XFree86 is modular (in so far as modules can be loaded at start up), and the X protocol itself is extensible. This has allowed X to continue to be used for many years after it was first released.

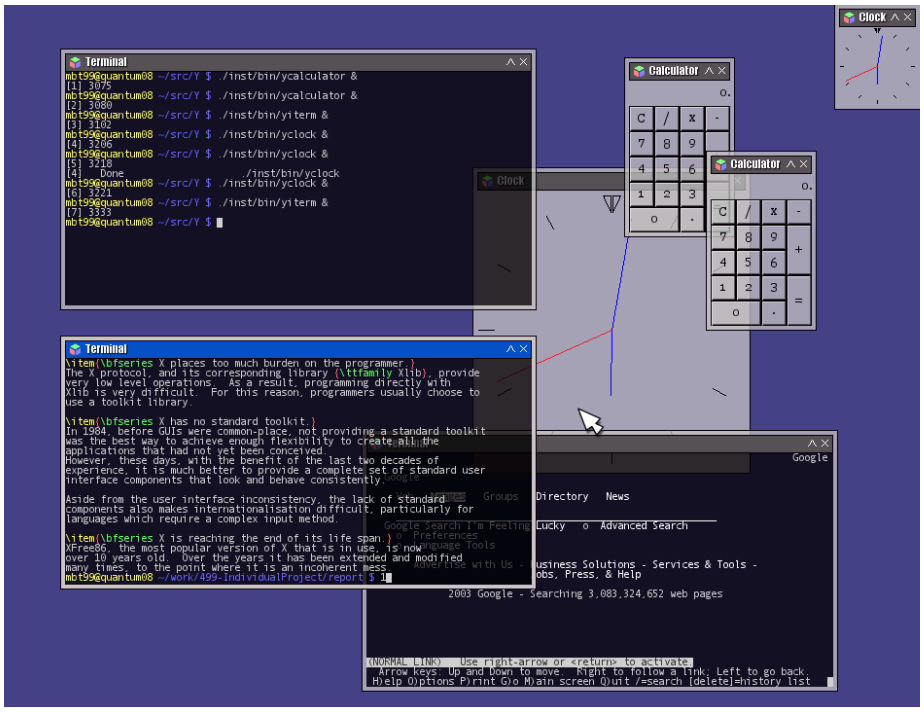
Clearly a full replacement of an entire graphical user interface system is far beyond the scope of a fourth year undergraduate individual project. The aim is instead to create a suitable foundation upon which a replacement can easily be built.

In particular, 'Y' provides:

- A complete object model suitable for graphical user interface components. Additional loadable modules may provide additional object classes, and the existence of an object class may be interrogated by clients at run-time.

- A message passing system for accessing these objects on systems that support some form of message passing, and a socket system for passing messages over networks, or on systems that support fast local sockets.

- The beginnings of a set of widgets with appropriate input event processing that can be used to easily and quickly build consistent applications.



```
mbt99@quantum08 ~/src/Y $ ./inst/bin/ycalculator &
[1] 3075
mbt99@quantum08 ~/src/Y $ ./inst/bin/ycalculator &
[2] 3080
mbt99@quantum08 ~/src/Y $ ./inst/bin/yiterm &
[3] 3102
mbt99@quantum08 ~/src/Y $ ./inst/bin/yclock &
[4] 3206
mbt99@quantum08 ~/src/Y $ ./inst/bin/yclock &
[5] 3218
[4]   Done                    ./inst/bin/yclock
mbt99@quantum08 ~/src/Y $ ./inst/bin/yclock &
[6] 3221
mbt99@quantum08 ~/src/Y $ ./inst/bin/yiterm &
[7] 3333
mbt99@quantum08 ~/src/Y $
```

```
\item{\bfseries X places too much burden on the programmer.}
The X protocol, and its corresponding library {\ttfamily Xlib}, provide
very low level operations.  As a result, programming directly with
Xlib is very difficult.  For this reason, programmers usually choose to
use a toolkit library.

\item{\bfseries X has no standard toolkit.}
In 1984, before GUIs were common-place, not providing a standard toolkit
was the best way to achieve enough flexibility to create all the
applications that had not yet been conceived.
However, these days, with the benefit of the last two decades of
experience, it is much better to provide a complete set of standard user
interface components that look and behave consistently.

Aside from the user interface inconsistency, the lack of standard
components also makes internationalisation difficult, particularly for
languages which require a complex input method.

\item{\bfseries X is reaching the end of its life span.}
XFree86, the most popular version of X that is in use, is now
over 10 years old.  Over the years it has been extended and modified
many times, to the point where it is an incoherent mess.
mbt99@quantum08 ~/work/499-IndividualProject/report $ 1
```

```
                                                              Google

                     Directory   News

                     Lucky   o  Advanced Search

                     usiness Solutions - Services & Tools -
                     obs, Press, & Help

         2003 Google - Searching 3,083,324,652 web pages

(NORMAL LINK)   Use right-arrow or <return> to activate.
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Figure 1.1: Current Implementation of Y

- A modular graphics rendering system that provides support for rendering the widgets in a variety of themes to a variety of graphics hardware. Unlike X, modules are unloadable and reloadable, which allows the display driver to be changed at run-time, for example to upgrade video card drivers, or to switch to a remote desktop server.

- A client library for writing Y applications in C++. This is a reference implementation from which libraries can be built for almost any other language.

- Some sample applications, most notably a clock, a simple calculator and a terminal emulator.

In chapter 2 the history of windowing systems and their corresponding graphical user interfaces is presented, followed by a description of the state of the art.

In chapter 3 the possible designs for a new windowing system are analysed, and a suitable design for Y is selected. This is followed up with a detailed design of the components in chapters 4 to 8. Details about how clients that use this windowing system are provided in chapter 9.

Finally, in chapter 10, the project is evaluated and tested, conclusions are drawn, and some examples of the many extensions to this project are listed.

Chapter 2

Background

2.1 A Brief History of Windowing Systems

In 1945, Dr. Vannevar Bush [4] theorised about a future device, which he called a “memex”, ‘in which an individual stores all his books, records, and communications, and which is mechanised so that it may be consulted with exceeding speed and flexibility.’ Bush’s memex would allow people to store graphical images of documents, browse them, and link them together.

This paper inspired Douglas Engelbart between 1962 and 1968 to write his NLS (oNLine System), which pioneered the use of a mouse (which he called an X-Y Position Indicator), and a keyboard to store, edit and hyperlink documents together. Part of NLS was the “windowed interface” that provided the views onto the documents.

Further, in 1963, MIT graduate student Ivan Sutherland created Sketchpad, a system which allowed the manipulation of graphical objects on a CRT screen. This idea was further adapted into the *Pygmalion* [5] programming system and part of the Smalltalk programming language.

Still, up to this point graphical user interfaces were primarily limited to a single application. The Alto system, which developed into the Star system, from Xerox’s Palo Alto Research Centre was the first to tie together all these ideas into something that resembles what we would now expect of a graphical user interface.

In December 1979, in exchange for some of Apple’s stock, XEROX allowed Steve Jobs, Steve Wozniak and a group of engineers from Apple to tour their research facility, take notes, and implement some of their ideas. Apple then went on to build the Apple Lisa, and from there the graphical user interface took off.

In 1983, Apple released their first Macintosh computer. The “Mac” interface is arguably the basis of all popular graphical user interfaces. The Macintosh interface had overlapping windows, whose contents could be scrolled around; icons to represent files and programs; and pull-down menus from the top of the screen.

The next notable windowing system to be released was that of Commodore’s Amiga. The Amiga windowing system, Intuition, in addition to the features of the Apple system, also provided a colour display, proportional scroll-bars, and

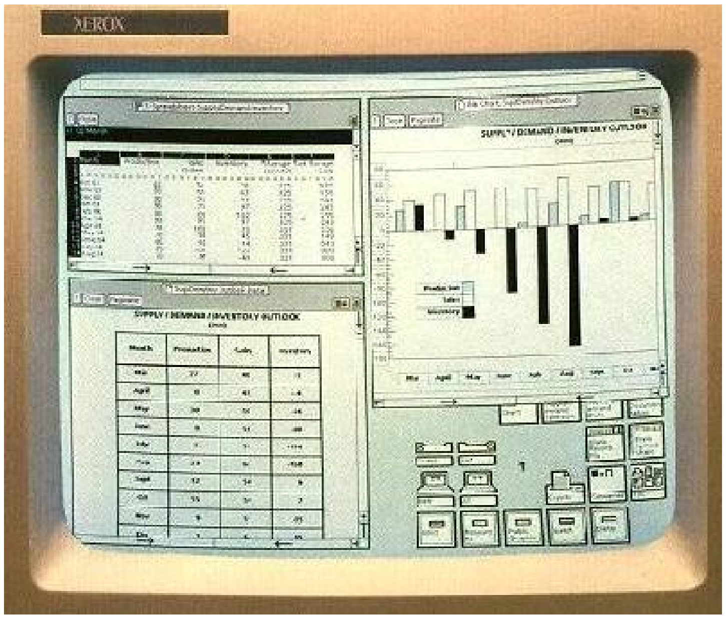


Figure 2.1: The XEROX Star User Interface

multimedia capabilities. It also had the concept of different virtual ‘screens’ upon which work could be placed. Switching between screens was achieved by clicking on an icon in the top right-hand corner of the screen.

In the mid 1980s, the X window system was developed at MIT. The major features of X were its network transparency and its extensibility. The network transparency allowed it to be used in main-frame style environments, where a large, powerful machine would run applications for many people, and a smaller, cheaper terminals connected to the mainframe by a fast network would run an X server for the applications to display their results on. Also, it is X’s extensibility that has meant X is still being used today. A typical installation of X on a machine running GNU/Linux uses around 26 of these extensions.

Meanwhile, Sun Microsystems developed their own windowing system, called NeWS. NeWS was also network transparent, but was based on a more powerful Display Postscript language. However, Sun’s unwillingness to share NeWS led to the adoption of X as the de facto standard.

Microsoft were fairly late entering the Graphical User Interface market, partly due to IBM’s belief that GUIs were a fad that would pass. Still, despite announcing Microsoft Windows in 1983, it did not arrive until 1985, and was missing several of the features promised, including the fundamental ability to overlap windows.

For many years Microsoft lagged behind in development, and generally copied the ideas of the larger players, particularly Apple. However, it gained significantly in popularity as the cheap IBM PC clones proliferated.

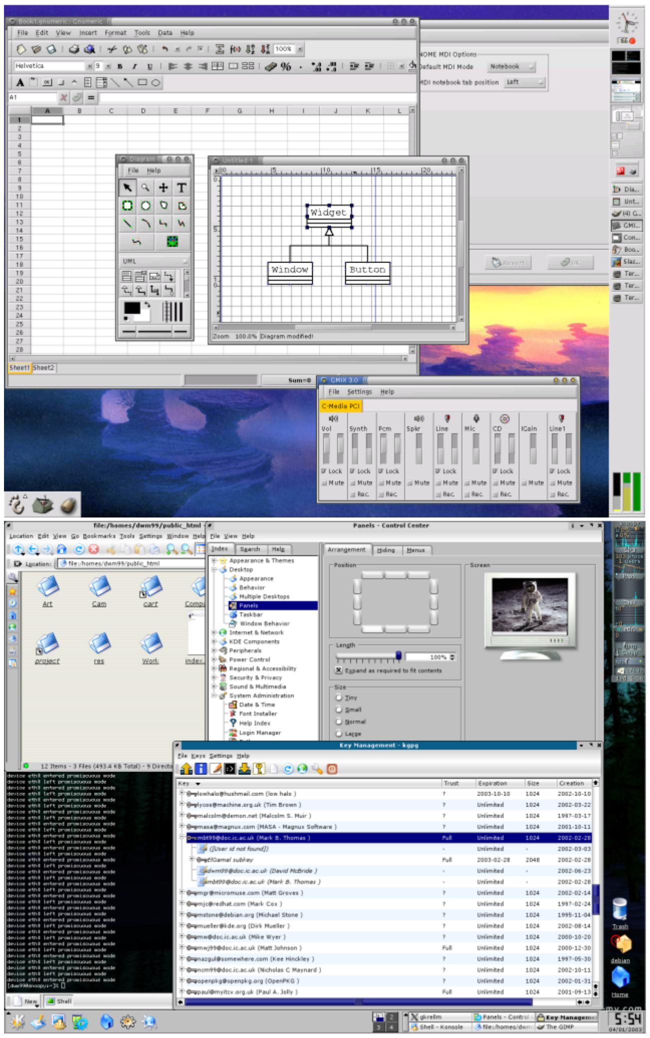
MacOS X’s windowing system, which is made up of the Quartz rendering engine and the Aqua interface, is the most recent windowing system. It added many modern features, such as anti-aliasing and alpha blending.

2.2 The State of the Art

The X Window System

The X window system is a very low level windowing system. It essentially provides overlapping windows, which are generally rectangular, although the XSHAPE extension allows them to be any shape.

X provides no standard GUI toolkit. Although the X Athena Widget set were commonly used in the early days of X, they were not sufficiently powerful for most applications' requirements. As a result Motif, GTK, Qt and a host of other incompatible widget sets were created, which has led to X becoming a user-interface nightmare. Some companies will not port their applications to X due to the lack of standard toolkit. Often GNU/Linux developers will write a command-line tool instead of a GUI tool for the same reason.



```
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
device eth0 left promiscuous mode
device eth0 entered promiscuous mode
[dwm99@snoopy:~]$
```

Key	Trust	Expiration	Size	Creation
lowhalo@hushmail.com (low halo)	?	2003-10-10	1024	2002-10-10
lycos@machine.org.uk (Tim Brown)	?	Unlimited	1024	2002-03-22
malcolm@demon.net (Malcolm S. Muir)	?	Unlimited	1024	1997-03-17
masa@magnux.com (MASA - Magnus Software)	?	Unlimited	1024	2001-10-11
mbt99@doc.ic.ac.uk (Mark B. Thomas)	Full	Unlimited	1024	2002-02-28
([User id not found])	-	Unlimited	-	2002-03-03
ElGamal subkey	Full	2003-02-28	2048	2002-02-28
dwm99@doc.ic.ac.uk (David McBride)	-	Unlimited	-	2002-06-23
mbt99@doc.ic.ac.uk (Mark B. Thomas)	-	Unlimited	-	2002-02-28
mgr@micromuse.com (Matt Groves)	?	Unlimited	1024	2002-02-14
mjc@redhat.com (Mark Cox)	?	Unlimited	1024	1997-02-24
mstone@debian.org (Michael Stone)	?	Unlimited	1024	1995-11-04
mueller@kde.org (Dirk Mueller)	?	Unlimited	1024	2002-08-14
mw@doc.ic.ac.uk (Mike Wyer)	?	Unlimited	1024	2000-10-20
mwj99@doc.ic.ac.uk (Matt Johnson)	Full	Unlimited	1024	2000-12-30
nazgul@somewhere.com (Kee Hinckley)	?	Unlimited	1024	1997-05-30
ncm99@doc.ic.ac.uk (Nicholas C Maynard)	?	Unlimited	1024	2002-10-11
openpkg@openpkg.org (OpenPKG)	?	Unlimited	1024	2002-01-31
paul@myitcv.org.uk (Paul A. Jolly)	Full	Unlimited	1024	2001-09-13

Figure 2.2: Screenshots of X using (top) GNOME 1.4 and (bottom) KDE 3.1

X's lack of standard GUI has led several groups of people to try and write coherent systems on top of X. The most notable ones are GNOME [12] and KDE [15], and these are illustrated in figure 2.2. Unfortunately, these all have one major problem: applications from one are not consistent with applications from another, nor the myriad of other X applications that people often use. Figure 2.3 shows several of the commonly used applications that have completely different interfaces, both in their look and behaviour.

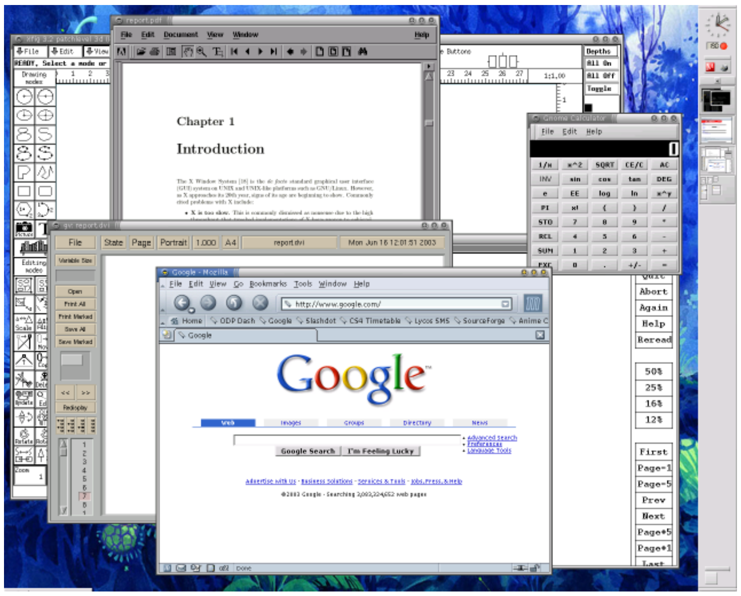


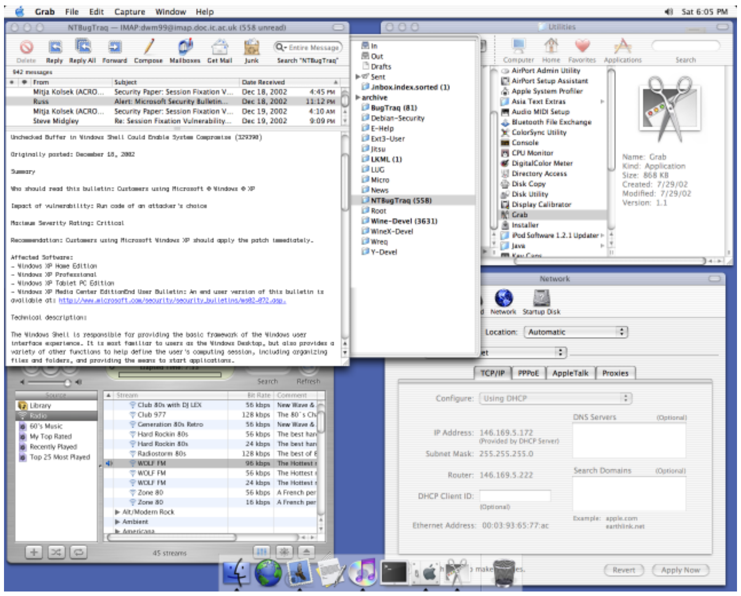
Figure 2.3: Various X applications and their differing interfaces

Windows are always opaque. Windows are unbuffered, which means that all drawing operations are sent directly to the screen's framebuffer. This means that they must all be clipped to the window's viewable area (which can be computationally expensive for complex graphics rendered into complex shapes), and whenever a window is partly uncovered, the application that owns that window must be contacted to repaint its contents within that region.

X is centred around a set of protocol specifications, most prominently the X protocol and the ICCCM (Ice Cubed) protocol. The X protocol is for applications and the window manager to communicate with the X server. The ICCCM protocol is for the applications to talk to the window manager. Recently, some desktop environments have begun adding extra protocols to allow applications to talk to each other, notably KDE's DCOP, and GNOME's use of CORBA.

X is very extensible, as extensions can be added to the protocol with relative ease. This is helped by the fact that most X implementations, particularly XFree86, are modular. Modules of extra code, for example device drivers, can be loaded into the server at start-up. The X internals provide sufficient hooks to allow modules to do useful work.

The main advantage of X over other windowing systems is its network transparency, which allows applications that are running on one host to directly use the display on another host without needing access to a display on their local host. This is invaluable for server administrators, who generally do not want their servers wasting processor time and memory on a windowing system.



From	Subject	Date Received	
Mitja Kolsek (ACRO...	Security Paper: Session Fixation V...	Dec 18, 2002	4:45 PM
Russ	Alert: Microsoft Security Bulletin...	Dec 18, 2002	11:12 PM
Mitja Kolsek (ACRO...	Security Paper: Session Fixation V...	Dec 19, 2002	4:10 AM
Steve Midgley	Re: Session Fixation Vulnerability...	Dec 19, 2002	9:09 PM

```
Unchecked Buffer in Windows Shell Could Enable System Compromise (329390)

Originally posted: December 18, 2002

Summary

Who should read this bulletin: Customers using Microsoft ® Windows ® XP

Impact of vulnerability: Run code of an attacker's choice

Maximum Severity Rating: Critical

Recommendation: Customers using Microsoft Windows XP should apply the patch immediately.

Affected Software:
- Windows XP Home Edition
- Windows XP Professional
- Windows XP Tablet PC Edition
- Windows XP Media Center EditionEnd User Bulletin: An end user version of this bulletin is
available at: http://www.microsoft.com/security/security_bulletins/ms02-072.asp.

Technical description:

The Windows Shell is responsible for providing the basic framework of the Windows user
interface experience. It is most familiar to users as the Windows Desktop, but also provides a
variety of other functions to help define the user's computing session, including organizing
files and folders, and providing the means to start applications.
```

Name: Grab
Kind: Application
Size: 868 KB
Created: 7/29/02
Modified: 7/29/02
Version: 1.1

Location: Automatic
Configure: Using DHCP
IP Address: 146.169.5.172 (Provided by DHCP Server)
Subnet Mask: 255.255.255.0
Router: 146.169.5.222
DHCP Client ID: (Optional)
Ethernet Address: 00:03:93:65:77:ac
DNS Servers (Optional)
Search Domains (Optional)
Example: apple.com earthlink.net

Stream	Bit Rate	Comment
Club 80s with DJ LEX	56 kbps	New Wave &
Club 977	128 kbps	The 80's Ch
Generation 80s Retro	56 kbps	New Wave &
Hard Rockin 80s	56 kbps	The best har
Hard Rockin 80s	24 kbps	The best har
Radiostorm 80s	128 kbps	The best of 8
WOLF FM	96 kbps	The Hottest
WOLF FM	56 kbps	The Hottest
WOLF FM	24 kbps	The Hottest
Zone 80	56 kbps	A French per
Zone 80	16 kbps	A French per
Alt/Modern Rock		
Ambient		
Americana		

Figure 2.4: Screenshot of Apple's MacOS X and its Aqua widget toolkit

Apple MacOS X

Apple's MacOS X is the tenth and most recent revision of their operating system. It uses their Quartz rendering engine, and the Aqua user interface to render graphics. An example of the MacOS X interface is given in figure 2.4.

The Aqua interface is generally consistent. One of Apple's selling points for the Mac interface is that they have always gone for consistency. Generally they have succeeded; for example it is often said that "Mac users know that Command-S is always 'save'." Apple publish very detailed user interface guidelines [1], which all applications developers should follow.

Unfortunately, these guidelines do not cover every case, and are not always followed, which results in some inconsistency in the implementation.

Quartz allows arbitrary transformations to be made to drawing operations, and supports the rendering of PDF data to the screen natively. It also has support for anti-aliasing and alpha blending (transparency). This results in the MacOS interface looking very aesthetically pleasing. It also uses animation to help the user understand what is going on.

Although bindings exist for Java and other languages, the main language interface, "Cocoa", is only available in Objective C. This makes rapid application development using simpler languages like Perl or Python impossible, and reduces the number of developers that can work on MacOS X applications.

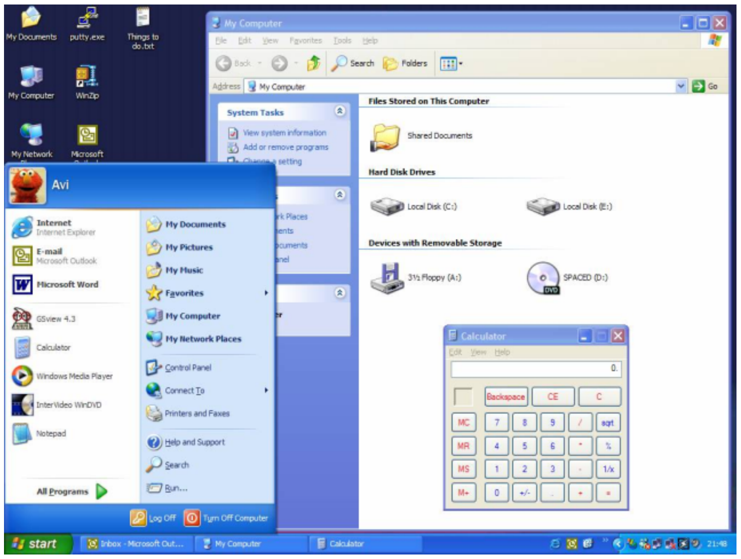


Figure 2.5: Screenshot of Microsoft's Windows XP "Luna" interface.

Microsoft Windows

Microsoft Windows XP is the most recent operating system from Microsoft. It features a brand new interface, which is themable. The standard theme is called "Luna", which is illustrated in figure 2.5.

Windows XP has many of the features of both X and MacOS listed above, though it has no outstanding features of its own.

A common criticism of Windows XP is its over-use of eye candy. Rather than using animation to explain to the user what is going on, it is sprinkled throughout the system for no good reason. Menus 'whoosh' open and bubbles pop up from window gadgets and system tray icons. Although these often are suitable for novice users, intermediate and expert users find them patronising and annoying as they distract them from their work.

Fresco

Fresco [11] also aims to be a successor to X, however their architecture is radically different. Fresco requires the use of CORBA as the communication primitive, which is intended to make network transparency and language independence easy.

In practice, Fresco has received little support. Most developers balk at the use of CORBA because its IDL specifications are arcane, and its APIs are difficult to use in any language.

Fresco also has a reputation of being slow. This may be due to it using InterViews as its GUI component system, or because it uses CORBA for communication. Most CORBA implementations are synchronous, which requires a round trip for every operation performed. Though this is partly alleviated by the high-level nature of the protocol, it will still cause problems for applications that wish to draw to canvases. A graphics application that wants to render a vector-based image with 1000 components will require thousands of round trips to the server. One of the reasons X was an improvement over the other contemporary UNIX windowing systems of the early 1980's was its asynchronicity [23].

Despite seven years of development, Fresco has yet to achieve anything more than rudimentary applications and a rough GUI.

DirectFB

DirectFB [9] is an abstraction layer on top of the Linux kernel framebuffer device, and provides rudimentary windowing system functions like those of X. DirectFB was designed for use on Embedded Linux devices and that is where it excels.

Recently, people have begun attempting to use it as a replacement for X. Other than being cleaner in design, DirectFB provides no advantages over X, and does away with its network transparency and several other nice features. DirectFB does not even provide multiple client support, and in order to simulate this, processes engage in co-operative multitasking, similar to that used by Microsoft Windows versions before 95. DirectFB still doesn't provide a standard GUI, though most DirectFB applications use GTK.

PicoGUI

PicoGUI [19] is also designed for embedded devices, though it goes several steps further than DirectFB. It uses a client/server model like X, but uses a bespoke, very light-weight protocol. It communicates in terms of server widgets which are identified by unique integers.

The PicoGUI developers have very recently decided to extend PicoGUI to become a replacement for X similar to Y. Where this will go remains to be seen.

Chapter 3

Server Overview

The design of the Y server is the most important part of the project, as it is this that dictates how clients will interface to it. Ideally the server should provide:

- **An advanced object system.** In addition to methods, fields, inheritance and overriding, the object system should provide facilities for advanced object orientation principles such as get and set modifiers on fields and loosely coupled signals. These features can then be implemented directly in client languages that support them, and indirectly in other languages, such as C++.

- **Language Independence.** A windowing system should not dictate the languages a client program should be written in. In order to allow many applications for Y to be developed, and to allow programs to be written in future languages, Y should make it easy for many languages to interface to it.

- **Various levels of access.** Different languages and different environments provide different communication primitives. High level object-based languages might best interface with Y at the object level, utilising their own object communication system. At a lower level, systems that provide fast message passing should be able to use this as their primary communications primitive. Finally, it should be possible to communicate with the server using socket connections, as this is the level at which network transparent applications will be able to communicate with the server.

- **Low dependency on the clients' capabilities.** Unlike X, where a lot of the computation is pushed on to the client, the Y server will be more "intelligent", and will be less dependent on the client. It should therefore be unnecessary for the client to implement "refresh" handlers that simply re-send what they have already sent.

- **Modularity and extensibility.** Using a plug-in system similar to that of XMMS [30], it should be possible to dynamically add code to the Y server. Ideally, modules should be unloadable and reloadable, so it is possible to switch from, say, one video driver to another on the fly. There should be reasonable hooks into the main server so modules can add real functionality to the server.

- **Multiple monitor support.** It should be possible to stretch the Y desktop across multiple monitors in a way that is transparent to client applications. Also, there should be support for multiple monitors displaying the same output, or partially overlapped output. It should also be possible for one monitor to follow the mouse as it navigates across a larger desktop area.
- **Support for Hardware Acceleration.** Whilst I do not expect the initial implementation to have any form of hardware acceleration, Y should be designed with it in mind. It should be easy for hardware vendors and other third parties to write a video driver for a particular piece of hardware and have it integrated with the server. The API for video drivers should be as stable as possible so that old device drivers do not have to be heavily maintained.

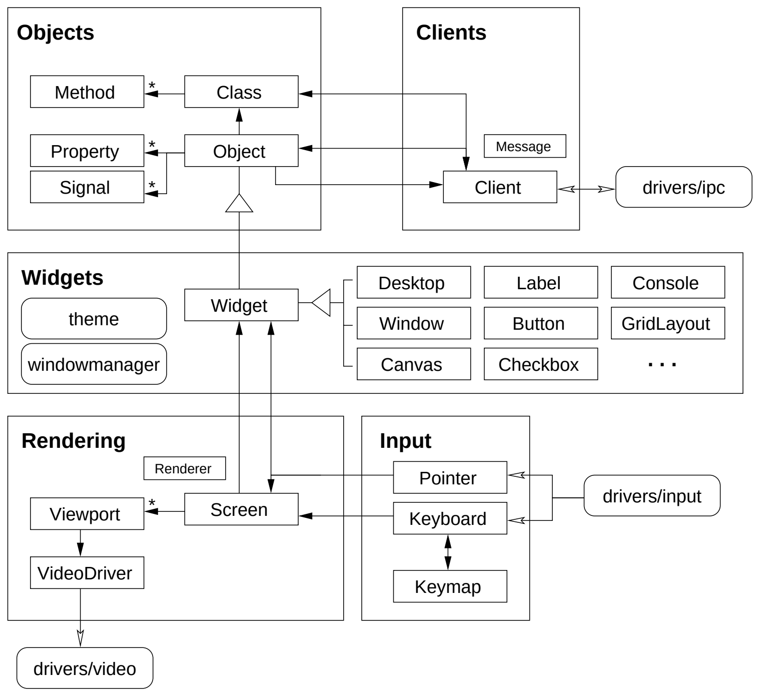


Figure 3.1: The Design of the Y Server

The design for Y is presented in figure 3.1. These components will be discussed further below, and in the following chapters.

3.1 Object Orientation

The object model for Y must be able to interface to many languages over several different communication channels. For this reason, it will be necessary to implement the object orientation internally. While it is tempting to use C++ as the internal language, there are several problems with this approach.

First, in order to provide remote access over simple communication channels, such as those based on message passing, it will be necessary for each class to provide a virtual table of method-name to function pointer mappings, for example:

```
canvasTable = {
 { "drawLine",      addr_of_canvas_DrawLine_function      },
 { "drawRectangle", addr_of_canvas_DrawRectangle_function },
 ...
};
```

To invoke a method, the interprocess-communication layer will search this table (if the table is kept sorted, then a binary search can be employed for $O(\log_2 n)$ look-up time), find the appropriate function pointer, and call it on the object that the call pertains to.

Unfortunately, this is not possible to express within the constraints of the C++ type system: pointers to members cannot be subsumed under the type of their receiver [27], that is a member pointer '`&Canvas::drawLine`' which is of type '`void Canvas::* (...)`' cannot be safely converted to a pointer of type '`void Object::* (...)`' even though '`Canvas`' inherits from '`Object`'. This restriction only applies to receiver types; function pointers can be subsumed under the basis of their return or argument types.

Second, interfacing modules to C++ is difficult. This is due to how C++ is based on C. C++ identifiers are 'mangled' into C identifiers in order for classes to have the same functions, and to allow polymorphism. For example, a method with the C++ signature '`void Canvas::drawLine (int, int, int, int)`' might be mangled to '`_ZN6Canvas8drawLineEiiii`'. This is not helped by the fact that the mangling format depends on which version of which compiler the module is compiled with.

Third, C++ does not provide some of the object orientation features that are desirable for the implementation of a window system. It does not provide get and set modifiers on fields, and it does not provide a loosely coupled event signalling system. Both of these can be implemented using large numbers of template classes, but this causes code bloat as new versions of the same class definition have to be created for each concrete type that the template is used for.

For these reasons, it is better to implement the desired features directly. The Y server will therefore be written in C. Modules can then be written in any language that can compile to a C-compatible shared object, and client libraries can be written in any language at all, provided that an interface layer is created for that language.

3.2 Standard Widgets

Y should eventually implement the large number of standard widgets that exist in almost every toolkit. Though this will not be necessary for the project, as a few widgets will suffice as a proof of concept.

Y should make it easy for widgets to be added to the server, and for clients to cleanly find out whether a widget is available. This will allow more complex applications to determine whether they should use a simpler widget, or perhaps even implement the widget on the client-side using a canvas on the server.

Finally, widgets should be themable, so that users can select the look and feel they desire. The theme should be provided in such a way that new widgets can leverage existing component looks to ensure that they fit within the theme.

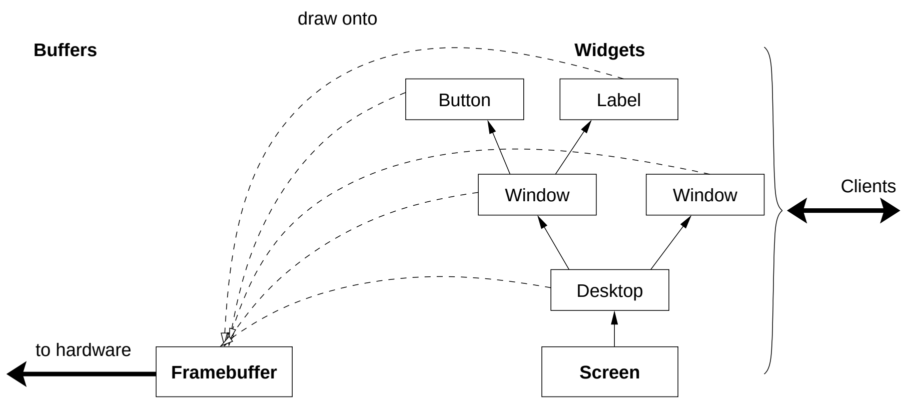
3.3 Buffer Arrangement

One of the characteristics of a windowing system is how it buffers widgets. Widget buffering is when the on-screen representation of the widget is stored in an off-screen buffer. Painting the widget from its abstract representation and rendering it to the screen are thus separated, which means complex widgets can be rendered quickly several times if they do not change. They can even be moved, covered and revealed without requiring a repaint. Buffering also enables one of the latest improvements in user interaction: alpha blending [20]. This allows widgets to appear partially transparent, which has great advantages in the efficient presentation of large amounts of information [31, 13]

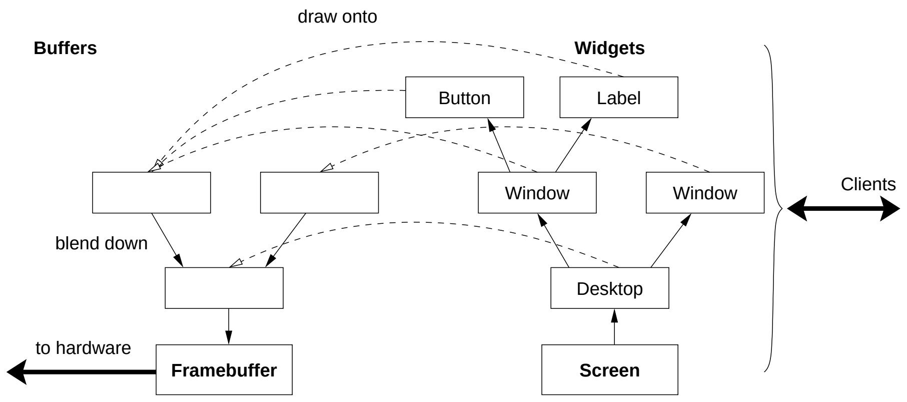
The chief disadvantage of buffering is the amount of memory it requires. If we are to buffer every widget, as shown in figure 3.2(c), complex applications will quickly exceed the available memory of an average machine. It is for this reason that older windowing systems such as X do not buffer any of their windows, as in figure 3.2(a).

With the recent increases of computer memory, however, we can afford to move to the more desirable "semi-buffering" model, as shown in figure 3.2(b). In this model top-level widgets, such as windows and pop-up menus are given their own buffers. This means moving, covering and uncovering windows that are unchanging will not require a repaint of the window and its contents. Further, widgets which are reliant upon the client for their exact contents, such as canvases, can also be buffered so that they do not require repainting whenever some other part of the window changes.

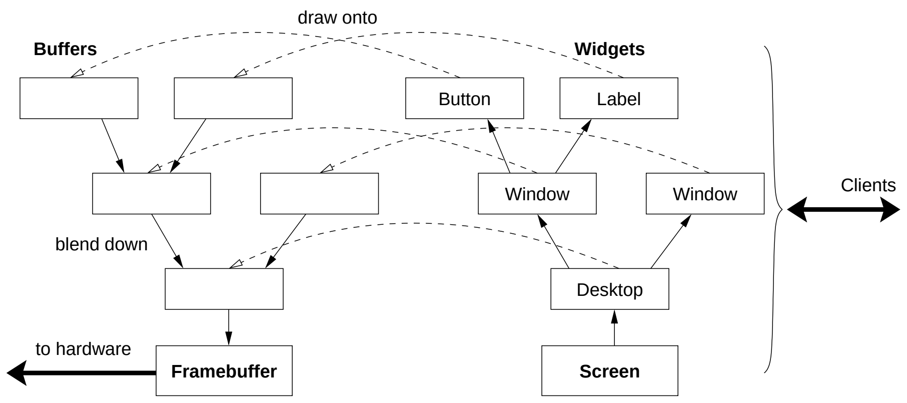
All other widgets paint themselves on to their container's buffer. This leads to a *paint-render* cycle, where all the widgets that have changed are painted on to the appropriate buffer, then all the buffers are rendered on to the frame-buffer.



(a)



(b)



(c)

Figure 3.2: Various Ways of Buffering Widgets

Chapter 4

Object Model

The Y server will require an object model that includes inheritance, method calls, overriding in sub-classes, and properties with get and set modifiers. It is also necessary for objects to be accessible remotely, and there should be no burden on the remote clients to maintain binary compatibility, that is there should be no 'magic' offsets to properties or methods.

Clients should also be able to subscribe to signals that can be emitted by objects, as these will form the basis of the event system.

In Y, both classes and objects are represented by data structures. A UML diagram for this is presented in figure 4.1

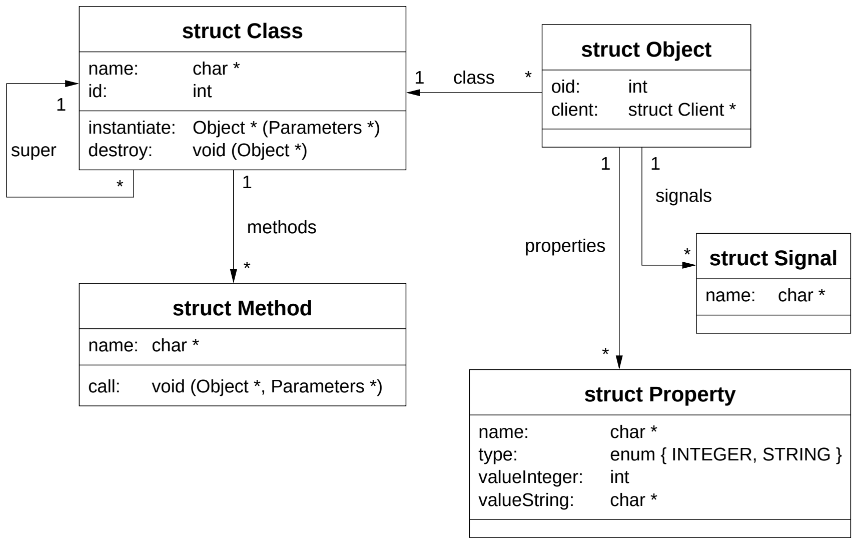


Figure 4.1: UML Diagram of the Object Model

4.1 Classes

Each class has a name, an `instantiate` function, a `destroy` function, and an array of method specifications. Each method specification consists of a mapping from a string name to a function pointer. Each class also has a pointer to its parent class, which may be `NULL` if the class has no parent.

Each class also has a unique numerical identifier in order to facilitate very fast retrieval of a class. This is not necessarily the same across all instances of the server, as they are assigned in order of discovery by the server. For this reason, functions are provided to find the numerical identifier from the class name.

4.2 Objects

Each object has a unique numerical identifier, a pointer to the client that it belongs to, a pointer to the class that it is a concrete instance of, and an indexed collection of its properties and signal subscriptions. An object instance may be found from its numerical identifier.

4.3 Instantiation and Destruction

If the `instantiate` function pointer in the class is not `NULL`, then clients may create instances of this object. Clients should call the `destroy` method when they are finished with an object. When a client exits, all of its objects are destroyed automatically.

4.4 Methods

A remotely invokable method is essentially a function with the signature:

```
void method (struct Object *object, struct Parameters *params);
```

The `object` is the object that the method is being invoked upon. If the method is being invoked without an object (`static` methods in C++ and Java), then this pointer is `NULL`. The `params` is a general purpose data type that allows the storage and retrieval of an arbitrary number of input and output parameters, which may be either strings or integers.

It might be possible at some point in the future to adapt method specifications to include type information about the parameters. Considering that methods may take various types of parameters (i.e. they may be polymorphic), and that well written methods should check the values that they receive anyway since they are coming from clients which may be untrusted, it is sufficient for now to require that methods check the parameters they receive for the correct number and type. Also the use of the `struct Parameters` gives us another advantage: it can neatly encapsulate multiple return values and error conditions.

To invoke a method on an object, the concrete class of the object is found by dereferencing the '`class`' field of the object. The method table for this class is then searched for the method name. If it is found, then the method is called with the appropriate parameters. If it is not found, then the '`super`' field of the class is dereferenced to move up to the super-class and the process is repeated until either the method is found, or a class with no super-class is reached. If the method is not found, then an error is returned to the client.

4.5 Properties

A property is a mapping from a name to either a string or integer value, and supports coercion from one type into another. Additionally, properties have function pointers for functions to be called before they are read, or after they are written to. These are known as get and set modifiers.

The set of properties an object has is indexed using a red-black tree to allow fast, that is $O(\log_2 n)$ access, insertion, and removal. Objects provide accessor methods to directly access their properties.

4.6 Signals

A signal is specified by its name, which is a string. Currently, only the client that owns an object may subscribe to its signals, and that subscription is represented by the existence of the signal's name in the object's signal index.

In the future, it may be desirable to allow different clients, or perhaps even other objects, to subscribe to the signals of an object. In this case, signals will need to be changed so that they contain a list of actions that should be performed when the signal is emitted.

A signal emission also includes a `struct Parameters` to carry any values that are associated with the signal; for example, a slider widget may include its updated value as a parameter to the "updated" signal. Objects provide a convenience function to emit a signal.

4.7 Messages

The message layer for Y is a protocol for accessing Y server objects over a system which uses message passing as its communication primitive. A message is a contiguous block of data that may be sent from one process to another, possibly over a network.

The Y message specification specifies the format of Y messages, and the possible operations that may be performed by a message. The specification also details how the message passing protocol may be implemented atop a stream protocol, such as that of UNIX domain sockets, or TCP/IP.

4.8 Message Format

Each Y message begins with eight 32-bit big endian unsigned integers, which are designated 'to', 'from', 'cid', 'oid', 'op', 'meta', 'ext' and 'len'. This is followed by 'len' bytes of data as the message body. The total message length is thus 'len' + 32 bytes.

The message fields are used as following:

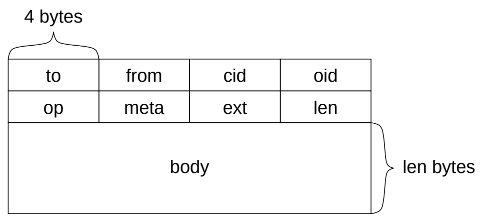


Figure 4.2: Message Format

‘to’	Indicates the ID of the client for which the message is intended, or ‘0’ if the message is intended for the server.
‘from’	Indicates the ID of the client from which the message came. The server will ensure that this value is correct.
‘cid’	Indicates the ID of the class of object for which this message is destined, or ‘0’ if no class is involved.
‘oid’	Indicates the ID of the object for which this message is destined, or ‘0’ if no object is involved.
‘op’	The operation code of this message. These are listed below.
‘meta’	Special information dependent upon which operation is specified by the ‘op’ field.
‘ext’	Reserved for future use. Should always be initialised to ‘0’.
‘len’	The length of the body that follows.

The format of the message body depends on the operation. A common body format is ‘delimited strings’. These are a sequence of one or more ASCII strings, separated by the ASCII Field Separator (`0x1C`).

Currently defined operations are:

`NO_OPERATION`	Performs no operation. Can be used for session keep-alive messages.
`ERROR`	Indicates an error condition to the client. The message body contains delimited strings detailing the precise error that occurred.
`QUIT`	Indicates to the server that the client is quitting.
`EVENT`	Communicates an emitted signal from the server to the client. The body contains delimited strings indicating the signal name and its parameters.
`INVOKE_SPECIAL`	Invokes a special method on the server. The body contains delimited strings indicating the special component, method name and parameters for the method invocation. The ‘meta’ field has its least significant bit set if the method is expected to return a value. See section 8.3 on page 40 for more details.

`SPECIAL_RETURNS`	Contains the return value of a special invocation.
`FIND_CLASS`	Requests that the server find the ID of a class from its name. The body of the message contains the class name.
`FOUND_CLASS`	Returns the ID of the requested class in the 'cid' field.
`INSTANTIATE`	Requests that the server instantiates the class whose ID is specified in the 'cid' field. The message body contains delimited strings indicating the parameters to the instantiation.
`NEW_OBJECT`	Returns the new object ID in the 'oid' field.
`INVOKE_METHOD`	Requests the server to invoke a method on the object specified in the 'oid' field. The method name and its parameters are given as delimited strings in the message body. The 'meta' field has its least significant bit set if the method is expected to return a value.
`METHOD_RETURNS`	Contains the return values of a method invocation as delimited strings.
`GET_PROPERTY`	Requests the server to return the current property value of the object specified in the 'oid' field. The name of the property is given as the message body.
`GOT_PROPERTY`	Contains the value of the requested property in the body.
`SET_PROPERTY`	Requests the server to set the property of the object specified in the 'oid' field. The property name and new value are specified as delimited strings in the message body.
`SUBSCRIBE_SIGNAL`	Requests the server to subscribe the client to a signal of the object specified in the 'oid' field. The signal name is given in the body of the message.

The message operation codes above `0x1000` are designated for special use by objects to be used for transfer of other data such as bitmaps.

4.9 Stream Implementation

Implementation of message passing over a two way stream is a simple case of serialising the message data as it is sent. Care must be taken to ensure that messages are not interleaved by threads sharing a connection.

Speed improvements can be made by buffering multiple requests and sending them as a single packet. However, the buffer must be flushed frequently in order to avoid significant latency. It must also be flushed whenever the messaging system tries to receive a message.

Chapter 5

Widgets and Themes

The widget hierarchy forms the main part of the Y server, and is its chief distinction from X. Rather than the X model, where widgets are implemented by various client-side libraries, widgets in Y are server-based objects.

There are about 40 different types of widget that might possibly be implemented, ranging from buttons to scrollbars to tree views. Since developing and testing widgets takes a large amount of time, the initial implementation will only include eight as a proof of concept.

5.1 Widget

The widget class is the super-class of all other widgets. It contains the general properties and methods of all widgets.

Every widget contains a parent-relative position and size. The position is parent relative to avoid updating all the child widgets whenever the parent moves. This is particularly useful for moving windows, as they are buffered, and so their children will not even require re-painting. Widgets also provide methods for converting local co-ordinates to global (screen-based) co-ordinates should that be necessary (for example to initiate pop-ups next to the widget).

Widgets also contain a set of size constraints, that is, their minimum, maximum and requested width and height. The widgets' size calculation routines guarantee that a widget will never be given a region less than their minimum size unless there is no way that can be achieved, for example if the minimum size is larger than the screen dimensions. The widget will also never be given more size than its maximum size, and will be automatically padded by the containing widget if there is too much space. In the absence of any other constraints, the widget will be given its requested size. The requested size should be the minimum amount the widget requires to display all of its information usefully, for example without scrolling, and no more.

For when the size constraints of a widget changes, the widget class also provides functions for propagating these changes up the widget tree, and if necessary rearranging the widgets. The widget changes its size constraint values, and calls `widgetReconfigure` on its containing widget. This then recalculates its size constraints, and calls `widgetReconfigure` on its container. This continues up the tree until either the top-most widget of the tree or a widget whose constraints are not affected by the change is reached (for example a fixed layout widget). This widget then calls `widgetResize` on its children, who then proceed to resize their children according to the size that they have available.

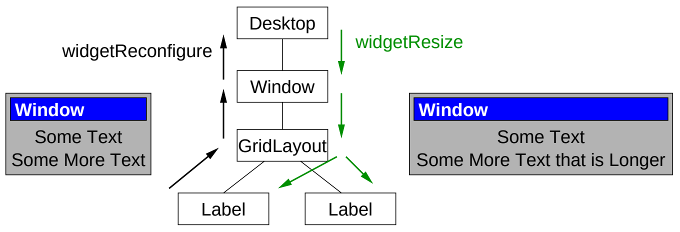


Figure 5.1: Widget Reconfiguration and Automatic Resize

For example: a window contains a grid layout with two labels one above the other, both with centred text. The window is sized such that the labels just fit within the window's border. The client application changes the text of one of the labels so it is now longer than it was before. The label recalculates its minimum width to be the minimum width of the text in the label, then calls `widgetReconfigure` on the grid layout. The grid layout then recalculates its minimum width based on the maximum minimum width of its children. Since this value is now larger, it calls `widgetReconfigure` on its container, the window. The window adds the size of the window border to the minimum width of the grid layout and sets this as its minimum width, and calls `widgetReconfigure` on the desktop. The desktop's minimum width is not affected by the windows that it contains, so it simply resizes the window so that its size constraints are satisfied, which in turn resizes the grid layout, which in turn resizes both of its children. This is shown in figure 5.1.

5.2 Desktop

The desktop is designed to be the root widget for most window managers. It represents a background, which is up to the desktop as to what it contains, and a collection of overlapping windows in a mutable z-order.

Since the desktop is designed to be used by the window manager, it currently provides no remote methods, properties or signals. It provides in-server functions for adding a window, raising a window to the top of the z-order, removing a window, and cycling through the windows that are on screen.

The desktop is, however, responsible for ensuring that windows that are maximised fill the full available space, and that windows do not vanish off the side of the screen when the desktop is resized, for example through a screen resolution change.

It is intended that this class will be extended significantly in the future to allow objects, such as icons, to be placed on the desktop background, and to allow panels to be attached to the sides of the desktop in an intelligent manner.

5.3 Window

A window is the main widget that applications will use. When an application starts, it creates a window object within which to pack all of its other widgets. Windows themselves only allow one direct child widget; to pack more than one widget inside a window, the application programmer must use an appropriate layout widget, such at the grid layout widget detailed below.

Windows are heavily dependent upon the current theme for nearly all their implementation details. This results in windows being highly themable in both their look and behaviour, which will allow themes and window manager modules that behave very differently from mainstream window managers to be built.

Windows maintain which of their child widgets are currently focused. This allows keyboard input to be maintained from one window to a next. When a window is deselected, it stores which child widget currently has the focus internally. When the window is reselected, it restores that focus. This allows the expected behaviour that when people switch back to a window they were previously using they can continue where they left off.

Windows also handle interactive moving and resizing of themselves. This is initiated by either the theme or window manager calling `windowStartReshape`, and is concluded by calling `windowStopReshape`.

Windows currently implement `setChild`, `setFocused` and `show` remote methods to allow the client to set the contained child widget, set the currently focused widget, and show the window on the screen respectively. They also have a `title` property which specifies their on-screen title.

The plan is to extend windows so that they have concepts about the type of widgets that nearly all windows have associated with them, that is menu bars, status bars and tool bars. This will allow the theme and window manager to decide what should be done with these, rather than the application programmer. This will allow the user to specify through theme selection whether they want their menu bars and tool bars attached to the top of the screen, or embedded within the border of the window.

5.4 Label

The label is probably the simplest widget. It has two remote properties: `text` and `alignment`. It displays the text in the default theme font according to the alignment. Current valid alignment values are `"left"`, `"center"` and `"right"`. The default is left aligned. Text is centred vertically, although it might be a good idea to add this as a property at some point in the future. Further developments may include the ability to change the colour and font.

5.5 Canvas

The canvas allows simplistic rendering in terms of graphics primitives such as lines and rectangles. To begin with, the canvas will only provide a few basic remote methods to allow drawing simple primitives. More complex primitives, such as ovals and arbitrary polygons can be added at a later date.

The canvas is double buffered, so the client may draw whatever it likes on the back buffer, whilst the front buffer displays a consistent image to the user.

When the back buffer is complete, the client calls `swapBuffers`, and the buffers are swapped and the widget re-rendered. This results in very smooth transitions from one state to another, and there is no visible redraw of the sort that plagues the canvases of many of the existing windowing systems.

Another problem that can be solved elegantly with this new design is that of the interactive resize problem. Traditionally, interactive resizes were performed with an 'elastic band' indicator, and only once the new size had been decided was the canvas repainted at the new size. With the increase in computational power, users have come to expect the more natural situation where window contents are displayed whilst resizing. Unfortunately, where canvases are concerned, this introduces a problem: each pixel of resizing generates a new "window has resized" event which is sent to the application. This requires the applications to be intelligent, and anticipate receiving a series of resize requests and merge them into one event. Unfortunately, not all applications do this, and this leads to the undesirable lagging behind effect seen when resizing applications such as Netscape Navigator 4.

Y solves this problem by treating canvas resize notifications in a different way. A client application is only notified once when a window resizes, but it is not notified what that new size it. It must then request the new size (for convenience, this can be bundled together with a "reset the canvas to the background" request), and is then given the most up-to-date size. If the window continues to resize after this point, the process is repeated. This results in only one "resize" event notification being live at any one point, which solves the interactive resize problem without any intelligent calculation on the part of the client.

5.6 Button

Button widgets represent the usual command buttons on windows, such as "OK" and "Cancel" buttons in dialog boxes. They have a '`text`' property which indicates the text that is displayed on them, and they emit a '`clicked`' signal when they are clicked on.

Buttons grab the mouse whilst they are being clicked so that the user may cancel them by dragging off of them.

5.7 Checkbox

Checkboxes are the usual on/off toggle widget found in most GUIs. The appearance of the checked state depends on the theme, so it can be marked by a pressed button, a ticked box, a box with a cross in, or anything else.

Checkboxes have a text label, which usually appears to the right of the check area, and a '`value`' property which indicates the current state as either '0' (unchecked) or '1' (checked).

5.8 Console

The console is a specialised widget designed particularly for the terminal emulator, but may be used by anything that requires a console-like display. A console is an array of characters, and is manipulated by setting attributes (such as colour and style) and writing text strings to locations on the console specified by their row and column. The console can also scroll selected rows of the screen by a particular amount, or can clear a rectangle specified in terms of character rows and columns.

The operations that the console has are designed to match closely with the operations required by libiterm. This means the terminal emulator, which is implemented using libiterm, can be implemented with the minimal effort.

5.9 Grid Layout

The grid layout is the sample layout widget in Y. Children can be placed at particular grid locations, and be given widths and heights in terms of grid cells. The grid layout then calculates the appropriate cell width and height and fits its children to the grid. For now, the grid is homogeneous, in that all the column widths are the same, and all the row heights are the same. In the future, it may be a good idea to make the grid optionally heterogeneous, so that columns and rows can be different sizes to better adapt to the sizing needs of the children.

Chapter 6

Graphics Rendering

The graphics rendering system incorporates the buffers, painters, renderers, viewports and video drivers that convert the abstract representation of widgets into concrete on-screen elements.

The central component of graphics rendering is the 'screen'. The screen contains the root widget and the collection of viewports. It is the screen's responsibility to tie the two together.

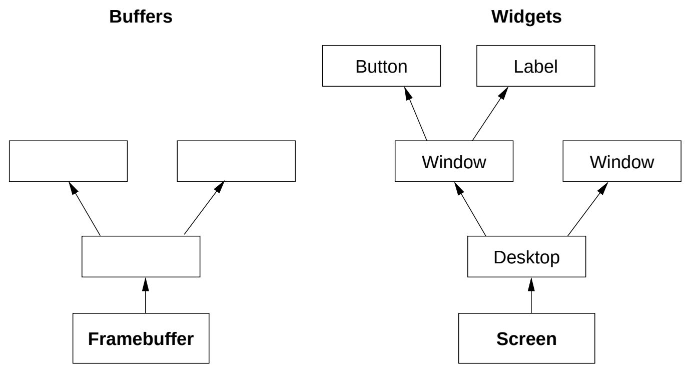


Figure 6.1: Initial Design of the Buffer System

The initial design for the Y buffer system involved the framebuffer maintaining a tree of buffers. This tree would essentially mirror the structure of the widget tree's buffered widgets. The widgets would manipulate the buffers through access procedures on the screen. This is detailed in figure 6.1.

In the initial prototype of Y, however, this design proved to have significant problems. First, it became difficult to define a clean API that would allow proper manipulation of the buffer tree from within the widget tree. At some levels, for example that of the desktop widget, the buffer tree needs to maintain an explicit and mutable z-ordering of the widgets.

Second, some widgets, like the canvas, require double buffering. Repeatedly inserting and removing a buffer from the buffer tree is messy and hence undesirable.

The solution is to associate each buffer with the widget that it pertains to.

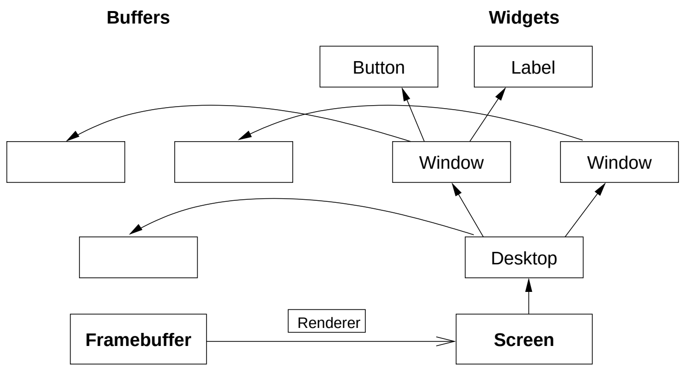


Figure 6.2: Final Design of the Buffer System

In essence, the widget ‘owns’ the buffer. Instead of iterating over the buffer tree, the framebuffer now creates a ‘Renderer’ visitor, and sends that to the screen which then passes it over the widget tree. Widgets with buffers render the appropriate buffer (if they have more than one) on to the renderer, and widgets without buffers do nothing. This is illustrated in figure 6.2.

6.1 Painters

Before the buffer is useful, it must first be filled with image data. This is done by a ‘painter’.

When a widget wants to update one of its buffers, it gets a painter from the appropriate buffer. It can then use this painter to paint its own image on to the buffer, and pass the painter to its children so that they may paint themselves on to the buffer.

The painter contains some state information about the current drawing operations, for example pen and fill colours, and provides some function pointers which allow drawing operations whose implementation is buffer-specific to be performed. The painter’s state may be saved on an internal stack, and then restored at a later time. This allows child widgets to modify the painter as they please, and then safely restore the painter to its original state before returning it to the parent.

Painters also provide translation and clipping mechanisms. Parent widgets can enforce the placement and size of a child widget by setting the painter’s clipping rectangle and origin offset to be the same as the child widget’s geometry. This process can be performed repeatedly, as it is expected to happen at each level in the widget tree. The painter can also be interrogated as to whether its clipping rectangle has entirely collapsed. This allows recursion to be stopped early when it is discovered than any child widgets will not be able to repaint because they have been clipped out by some ancestor.

The current painter implementation provides methods for clearing and drawing rectangles, and for drawing lines. Specially optimised functions for drawing horizontal and vertical lines are also provided. Functions are also provided for drawing a bitmap of alpha values on to the buffer with the current pen colour and for rendering RGBA data directly. These two functions are useful for font and icon painting.

6.2 Viewports

A viewport is the representation of one discrete rectangle of screen space. It corresponds to either a physical or, in the case of remote desktop applications, virtual monitor, and as such is associated with a video driver.

Note that some video drivers will be written for graphics adaptors which have multi-head support, so any one video driver may have any number of viewports associated with it. When a video driver module is loaded, it creates a viewport for each and every physical or virtual monitor that it has attached, and registers them with the screen. From this point, all interaction with the video driver will be done through the viewport. Video drivers are generally implemented in modules so that the appropriate one may be loaded into the server.

Screen updates in Y are driven by the viewport. This is because the viewport is the best component to know the appropriate refresh rate of the screen. Local video hardware drivers will want to update at least sixty times a second. Remote desktop drivers will almost certainly want to tailor their update rate to the available bandwidth.

Every time some widget in the screen changes, a rectangle indicating the area that changed is passed up the widget tree until it reaches the screen. The screen then passes the rectangle on to every viewport which stores a copy of it.

When a viewport determines that it is time to update, it collects together all these invalid rectangles and unions any rectangles that significantly overlap. It then creates a 'renderer' for each rectangle, and passes this to the screen, which passes it over the widget tree. This corresponds to the *render* part of the *paint-render* cycle.

In order to reduce the amount of redrawing required, buffered widgets can also store these invalid rectangles as they pass up the tree from their children. They can then wait until they receive a 'renderer' from their parent before they repaint their children on to their buffer. This causes some interleaving of the *paint-render* cycle which noticeably improves performance.

6.3 Fonts

Font rendering in Y uses the FreeType library [10] to allow access to several types of bitmap and outline fonts, particularly TrueType fonts.

FreeType only provides facilities for rastering characters and determining characters' geometries. The FreeType documentation gives examples on how to convert this into string rendering routines.

Y will require extra functionality from its font rendering system, as it will need to measure strings in order to determine widget sizes, and to correctly centre strings on screen. Measuring a string is almost identical to painting the string; the rasterised data is simply thrown away.

Text rasterisation, particularly from outline fonts, is a fairly computationally expensive operation. Since most strings which are measured will be subsequently painted, Y employs a string caching scheme. When a string is measured or painted, its rastered version is cached. If the string is then subsequently measured or painted again, the cache is used rather than re-rastering the string.

6.4 Renderers

Renderers are essentially tables of function pointers for video-driver specific functions. The current renderer API is designed with two types of renderer in mind. These are detailed below. As more video drivers are written and it becomes more apparent what is required of a renderer, appropriate functions can be added.

Software Renderers

Video drivers that have no form of hardware acceleration, or only accelerate opaque block transfers use software renderers. The software renderer blends all the data that it receives into an in-memory buffer that is the same size as the invalid rectangle. Once the data has been fully blended together, the block of data is transfered to the video driver so that it may copy it to the frame buffer.

Software renderers are the slowest, but most general type of renderer. A software renderer is used for both the SDL and Linux framebuffer device video drivers.

Accelerated Renderers

Graphics adapters that provide some kind of accelerated means of blending in-memory RGBA data to the screen will use accelerated renderers. Accelerated renderers pass the buffered data for each widget that is encountered directly to the video driver, so that it may blend the data itself using the accelerated method.

This method should be significantly faster than using software-based renderers, but will require video drivers which know how to perform the blending operation in hardware. Further, not all graphics adapters support blending data in this manner. They will also have to use software renderers.

Chapter 7

Input

The two main input methods for Y will be the keyboard and a pointing device, such as a mouse. Other input devices such as joysticks and graphics tablets will not be supported initially by Y, though it should be simple for them to be added in the future.

The Y server itself has no drivers for obtaining input events. Instead, it provides an abstract API which loadable modules can send the input events to by calling particular functions. This allows maximum flexibility for supporting various types of keyboards and pointing devices.

7.1 Pointing Devices

There are two types of pointing devices: relative and absolute. Relative pointing devices, such as a mouse, give offsets that indicate how far the device has moved since the last movement event. Absolute pointing devices, such as touch-sensitive screens, give the location within the screen that the pointing device is currently at.

In order to support both types of pointing device, Y provides two functions:

```
void pointerSetPosition (int x, int y);
void pointerMovePosition (int dx, int dy);
```

Both of these affect the same on-screen cursor, so it is possible to use both a touch screen and a mouse on the same Y server.

Pointers may also have an arbitrary number of buttons. In order to handle this, Y provides a function:

```
void pointerButtonChange (int button, int state);
```

which allows drivers to update the state of any button.

These events are passed immediately to the screen, which forwards them to the root widget. It is then the responsibility of the containment widgets to pass the events down through the containment hierarchy until the leaf widget that the pointer event pertains to is reached.

Certain widgets will require notification of events outside their bounding box for brief periods of time. This is usually associated with the user 'grabbing' the widget with the pointer button. The simplest example is when cancelling a button press. If the user drags the pointer out of the button's rectangle whilst keeping the pointer button pressed, the button reverts to its 'unpressed' look to indicate that the button press may be cancelled. The button widget will therefore need to know about pointer movement events outside its bounding box whilst the pointer button is pressed.

To facilitate this, Y provides two more functions:

```
void pointerGrab (struct Widget *);
void pointerRelease ();
```

The first causes the pointer events to be forwarded directly to the widget that requested them rather than sending them to the screen. The second cancels the previous grab and restores the sending of events to the screen.

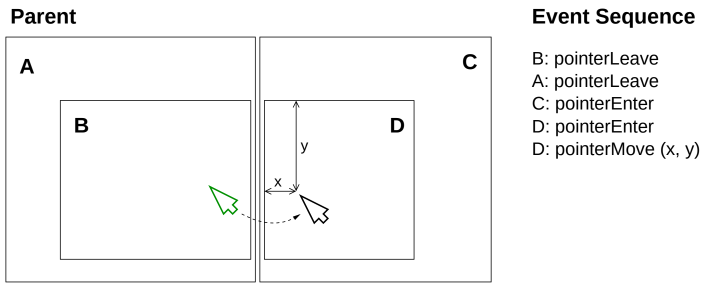


Figure 7.1: Pointer Enter and Leave Events

Within the widget hierarchy, it is desirable to have notification of when the pointer enters or leaves a widget. This allows widgets to become highlighted when the mouse hovers over them, which is useful for user feedback. Figure 7.1 shows an example of the events that we want to occur.

7.2 Keyboards

Similar to pointer devices, keyboard input is obtained by appropriate driver modules. The module passes keyboard input to Y itself by means of two functions:

```
void keyboardKeyDown (enum KeyCode code);
void keyboardKeyUp (enum KeyCode code);
```

The `KeyCode` enumeration contains codes for any key that may appear on any keyboard.

Unlike pointer events, keyboard events do not have an associated position, and so it makes little sense to try and send them through the widget hierarchy.

Instead, there are two functions:

```
void keyboardSetFocus (struct Widget *);
void keyboardRemoveFocus (struct Widget *);
```

which set and remove which widget is currently focused. Focus can be set either by mouse clicks on to certain widgets, or by certain key presses.

The only exception to this rule is keyboard shortcuts which correspond to special window manager commands. These are those that cycle through windows, switch workspace, or close a window. For increased consistency, Y specifies that one modifier set be assigned to window manager functions. For the current implementation, I am using the left and right 'Windows' keys.

Upon pressing either window manager key, the window manager is notified that its modifier is pressed. From this point, all keyboard input is directed to the window manager module, rather than to the focused widget. When the window manager key is released, it is again notified of this, and all further input is directed to the currently focused widget.

7.3 Keymaps

Additional problems are caused by the fact that the mapping from key to symbol varies from keyboard to keyboard, especially with respect to modified symbols.

For this reason, Y will provide a keymap system where keymap specifications which translate a keyboard key into a key symbol can be loaded into the server. For example, a keymap entry which binds the shift and control-alt states of the "4" key (keycode 52) to "$" (36) and "€" (164) respectively looks like:

```
keycode 52 = SHIFT(36), CTRL ALT(164)
```

Chapter 8

Miscellanea

8.1 Abstract Data Types

The Y server will require abstract data types for:

- **Lists of data.** These should be implemented as a doubly-linked list with references to both the start and the end node stored in the list header. It should be possible to iterate quickly over the list in both directions.
- **Priority queues of data.** These should also be linked lists, but the contents should be stored in a sequence defined by some comparison function provided to the queue at initialisation time.
- **Explicitly ordered data.** These should be similar to linked lists, but allow several methods of re-ordering the elements within the list; in particular moving elements up or down by one position, promoting elements to the top of the order, or demoting elements to the bottom of the order.
- **Indices of data.** These should provide fast look up of data based on a key, plus fast insertion and deletion. Indices can be implemented using any type of balanced tree, such as an AVL tree or a Red-Black tree. The current implementation uses a Red-Black tree implementation based on the one in GNU libavl [18].

8.2 Configuration

Configuration details are taken from a configuration file. The default location of this file is in the `etc/Y` directory inside the installation prefix. The default configuration file is named `default.conf`.

A different configuration file may be set by either setting the `YCONFIGFILE` environment variable to the full path of the configuration file, or by passing the `--config=/path/to/file.conf` option at Y start-up.

The configuration file is formatted as a sequence of commands to be passed to the various Y components. An example Y configuration file might be:

```
module load "theme/basic"
module load "wm/default"
module load "driver/video/fbdev" mode=1152x864-85
module load "driver/ipc/unix" socket=/tmp/.Y-0
module load "driver/ipc/tcp" port=8900
keymap load "gb"
```

8.3 Special Functions

Certain parts that are normally internal to Y will want to expose some sort of interface to clients for configuration and information reporting. The 'special' function interface provides a canonical way to do this.

Each part of Y that wishes to expose a special interface registers itself with the special interface index, providing a string name and a function pointer for a handler.

When a client sends a message of type `INVOKE_SPECIAL`, the appropriate special handler is sought, and if found the rest of the parameters are passed on to the handler. The handler may return some results if it wants to, and these are passed back to the client if they were requested.

The best example for this is that of viewports and their resolutions. Viewports will want to expose some interface to the clients so that clients can change screen resolution. However, since the clients cannot instantiate viewports, there is little point in them being full-blown remote objects.

Instead, the screen registers a '`viewport`' special. The handler that is associated with this accepts two possible inputs. The string '`list`' causes it to send back a list of the current viewports along with their numeric IDs. An integer that is a valid viewport ID causes it to pass the remainder of the parameters on to the viewport itself.

The viewport's handler allows the client to either '`listResolutions`' to obtain a list of valid resolution settings, or '`setResolution`' to change the viewport's resolution to a new setting.

Chapter 9

Clients

Clients communicate with Y using messages over some kind of socket-based stream, such as UNIX domain sockets, or TCP/IP. See section 4.7 for details on how the messaging system works.

Of course, application developers will not want to continuously construct, send, receive and interpret messages in order to write graphical applications. For this reason, client libraries for Y will need to be built.

9.1 Client Library Specifications

A client library must provide:

- A means for client applications to establish a connection to a Y server based on a server identification string. Server identification strings are of the form `<protocol>:<address>`. The format of the address part depends on the protocol. For UNIX domain sockets, the protocol is `unix` and the address is the path of the socket. For TCP/IP sockets, the protocol is `tcp` and the address is the host name or IP address of the host to connect to, followed optionally by a colon and port number. If the port number is omitted, it defaults to 8900, the default Y TCP/IP port.

- A means of querying the server for classes. Applications should be able to query whether certain classes exist, as this will form the basis of Y's extensibility.

- A means of sending special requests and receiving the replies. This will allow clients to perform special operations, such as changing viewport resolution. See section 8.3 for more details.

- Skeleton classes for each of the classes in the server that the client library knows about. A skeleton class provides the same interface as the remote class, and forwards all requests to the remote class. Client libraries can take advantage of the special features of some languages. For example, if a language supports get and set modifiers, then the client library can use these for object properties. If it does not, then these can be supported using the traditional `getProperty` and `setProperty` style methods.

- A main loop. Most GUI applications are callback based. This means that when an application has initialised itself, it passes execution control over to the client library. The client library then waits for events from the GUI server and calls callbacks when they occur. The main loop should be able to allow the client to listen on other file descriptors as well as the server connection, and to trigger events after a short delay.

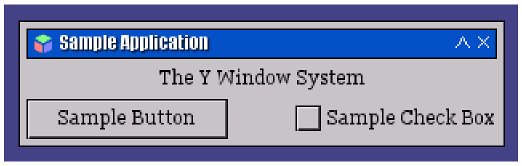
9.2 The C++ Client Library

For this project a client library for C++ has been built. To make it easier to write client libraries, the C++ library is split into two halves: libY and libYc++. LibY is written in C and provides generic functions to construct, send, receive and deconstruct messages, as well as establish a connection to the appropriate Y server over the appropriate protocol. This library can also be used by any other language binding for which interfacing to C libraries is simple.

9.3 Example Applications

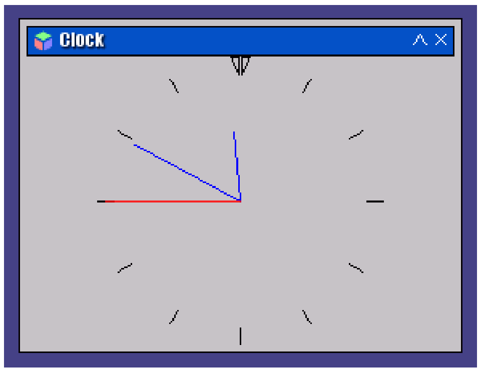
This project delivers four sample applications, plus a rudimentary control application ‘`yctl`’

`ysample`



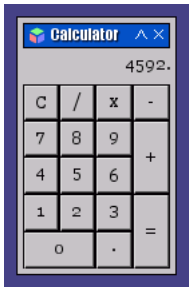
This is a basic sample application intended to show off Y’s standard widgets, and provide an example from which new Y developers can learn. It simply creates a window and packs a label, button and checkbox inside it.

`yclock`



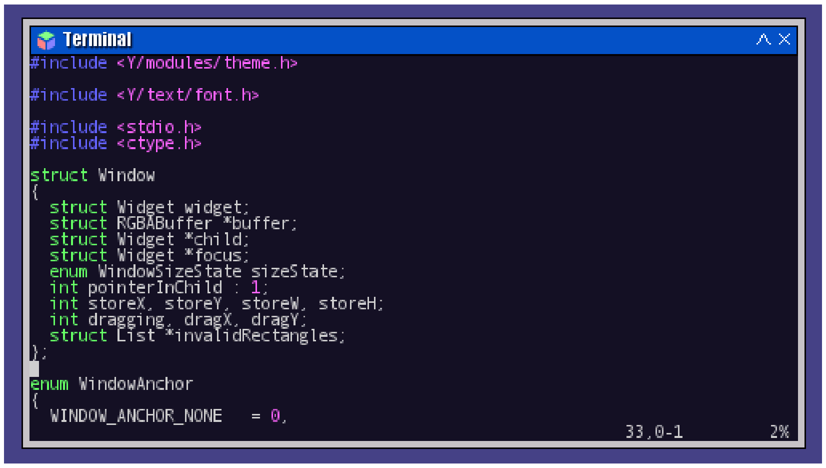
The clock is based on a canvas. Once per second, it re-draws a representation of an analogue clock onto the canvas.

`ycalculator`



The calculator is a simple stack-based calculator that uses button widgets for buttons and a label to display the result. The calculator is not particularly complex, and whilst capable of performing any simple calculation, it is not particularly robust with respect to erroneous entries.

`yiterm`



```
#include <Y/modules/theme.h>

#include <Y/text/font.h>

#include <stdio.h>
#include <ctype.h>

struct Window
{
  struct Widget widget;
  struct RGBABuffer *buffer;
  struct Widget *child;
  struct Widget *focus;
  enum WindowSizeState sizeState;
  int pointerInChild : 1;
  int storeX, storeY, storeW, storeH;
  int dragging, dragX, dragY;
  struct List *invalidRectangles;
};

enum WindowAnchor
{
  WINDOW_ANCHOR_NONE   = 0,
                                                              33,0-1          2%
```

The terminal is based on `libiterm` [26] by Jiro Sekiba, which provides a simple API to a generalised terminal emulator. The Y terminal emulator program simply creates a console widget and an instance of an iterm VT and connects the two together.

`yctl`

This small program provides a direct command line interface to the special interfaces in Y. It is intended as a development aid, as the final version of Y will have a graphical control centre which enables configuration of all the aspects of the Y server graphically. Since Y does not yet have a full set of widgets, it is not possible to start writing the control centre yet.

The `yctl` program simply passes its command line arguments along to the Y server as though they were a special request. This means the resolution of the first viewport, for example, can be changed from the command line using:

```
yctl viewport 0 setresolution 1024x768-60
```

Chapter 10

Conclusions

10.1 Evaluation

The aims of this project were to:

- Design the framework for a successor to the X windowing system;
- Implement the beginnings of this framework, up to the point where a few sample applications, such as a clock, calculator and terminal emulator can be built;
- Support all the useful features of X, in particular its network transparency and extensibility; and
- Improve on X, and enable extra features like widget buffering, alpha transparency, and desktop resizing.

These goals have been mostly achieved.

Certainly the project has delivered a simple yet powerful design for a very extensible window system and associated communication principles. By adding to the server a concept of GUI widgets, several improvements are made. First, all applications are forced to use the same set of widgets. This makes all widgets look and behave the same. It also opens up more possibilities in the realms of accessibility and internationalisation than the myriad of toolkits that were available for X. Second, the weight of the communication protocol between the client and the server is significantly reduced. Instead of the client repeatedly sending the hundred-or-so instructions it takes to draw a button in its various states, it simply requests a button object at a particular location; the server does the rest.

The implementation of the simple server is as complete as was expected. There is a complete object system which can be accessed remotely over a variety of mechanisms. The object system is language independent, and does not depend on any specific features of any language. The server is modular, and modules can be unloaded and reloaded at run-time with no adverse effects on the server. Support for advanced features such as multiple monitors and hardware acceleration has been considered in the design. With a small programming team filling in the few missing parts, a deployable version of Y could be produced in a matter of weeks.

Extensibility is at least as good as that in X. The Y protocol is nearly infinitely extensible, as any new extension that requires client communication can use an object class if it needs to hold data for clients, or a special interface if it does not. Modularity is significantly improved over existing X implementations as code can be safely *unloaded* from and *reloaded* into the server. This has important ramifications, as it may be possible to change graphics card driver version on-the-fly, or dynamically load a VNC server module to allow connection to the Y display from a remote machine.

Y also provides widget buffering, removing the need for client redraw, and enabling alpha transparency. By starting afresh, Y is able to provide desktop resizing (and potentially rotation) and other modern features without breaking anything else.

As it stands, Y still suffers from the same configuration problems that plague X. Y configuration is specified by a text file, and whilst this is the UNIX standard and is very useful for system administrators, it poses difficulty for the home desktop user. Ideally Y should manage its own configuration file, automatically detecting what hardware is available and setting sensible default values, but as yet it does not do this. It could also integrate with some other configuration system that allows users to set settings for all their applications. As yet, no such system exists in a usable form. This is definitely an area for improvement.

The current Y implementation is in no way tweaked for performance. In order to get the best out of stream based I/O, as much data should be sent in one go as is possible. Though it would not be hard to add into Y, the current implementation still sends data in small blocks. In particular, each Y message is sent in two separate parts: the header and the body. Improving this may improve Y's performance significantly.

Despite this, Y's network transparency is an order of magnitude faster than X's. Applications that are run on a local network are almost indistinguishable from applications on the local machine. Further, applications can now successfully be run on a machine that is some distance away *over the Internet.* For example, I have successfully run and used the calculator and clock examples on a Y display in the Computing lab at Imperial College in South Kensington, from my home computer in Ealing some 5 miles away, connected via ADSL to the Internet. The latency is comparable to the latency experienced when using X applications over a local area network. The network traffic caused by a simple Y application is negligible.

Y has no security model, and currently anyone can connect to any running server and display applications. Clearly this is undesirable. It is also undesirable to have a complex authentication system 'bolted on' to the Y server like the `MIT_MAGIC_COOKIE` system is to X. It is recommended that this issue is addressed before any release of Y is made.

In comparison to X, Y is a significant improvement. Since all widgets are implemented in the server, latency will always be minimal, even if the application itself has a high latency connection to the display server. Further, there is now a standard widget set which is fully themable, and more efficient to use.

Y's API is considerably simpler to use than X's. A simple application to open a window and close it when the user clicks the close gadget is given in appendix C on page 59. A comparable Xlib program would require almost two hundred lines to create establish the connection, create the GCs, create the window, map the window, enter an event dispatch loop, listen for expose events in order to draw the window's contents, and listen for the close request.

Refer to appendix C for the API documentation of the C++ client library.

By breaking clean from the X protocol and the current X implementations, Y has a clean design created using modern software engineering techniques. This makes it much easier to understand, maintain and modify. All of the problems that were associated with X can be solved with Y with relative ease, if they have not already been solved.

In comparison to Windows and MacOS X, Y is a capable competitor. Y is capable of eventually providing all the functionality that users of these two windowing systems expect. It will be as consistent as MacOS and as configurable as Windows.

In comparison to Fresco, Y is a better successor to X. Y is noticeably faster than Fresco, probably due to its simpler architecture. Y is less dependent on other software, such as CORBA ORBs and abstraction libraries. Y is easier to write applications for as it does not require learning the CORBA IDL, and the CORBA interface for your language of choice. Finally, Y is just as extensible as Fresco, as both allow dynamic loading of code into the server.

10.2 Testing

There are a variety of tests that can be performed on Y to test the various facets of its functionality.

Unit Testing

All the abstract data types used in Y are automatically tested by running '`make check`'. These automated tests create instances of the data structures, then perform thousands of random operations on them, checking that they maintain data integrity.

Further, the automated tests include some regression tests for the few bugs that were found in the implementations.

It might be desirable to extend the unit tests to incorporate other core features of the Y server, such as the object system. This might be hard to do in practice.

Usability Testing

In its current state, Y does not have enough features for useful usability testing to begin. Once more complex widgets have been created, it will be important to begin usability testing.

Performance Testing

Due to the current implementation not using hardware acceleration, Y's performance is not as good as it could be. It is still perfectly usable, but there is some noticeable slow down, particularly when blending large windows to the display.

Profiling the server using the GNU `gprof` program shows that the server is spending 84.69% of its running time in the `swrendererBlitRGBAData` function, that is the function that alpha-blends the buffered data to the display. This function can be entirely accelerated on supported hardware if suitable device drivers are written.

Y's performance really shines when applications are run over a network. The table below compares some figures for starting and using the Y calculator, the X calculator and the GNOME calculator. Operations are stop-watch timed from the moment the request was made to the moment when the screen had finished updating. Each operation was performed five times, and the mean time was taken. The results are presented in table 10.1

Operation	**Y**	**X**	**GNOME**
Local Machine			
Starting	0.1 s	0.1 s	0.3 s
Resizing	$<$ 0.1 s	0.1 s	0.1 s
Moving	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
Exposing	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
Clicking a button	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
100 Mbps Network			
Starting	0.6 s	0.3 s	0.3 s
Resizing	$<$ 0.1 s	0.1 s	0.3 s
Moving	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
Exposing	$<$ 0.1 s	0.1 s	0.1 s
Clicking a button	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
512 kbps Connection to the Internet			
Starting	2.2 s	3.8 s	21.1 s
Resizing	$<$ 0.1 s	2.1 s	20.4 s
Moving	$<$ 0.1 s	$<$ 0.1 s	$<$ 0.1 s
Exposing	$<$ 0.1 s	0.5 s	24.2 s
Clicking a button	0.3 s	0.5 s	1.4 s

Table 10.1: Comparative latency over different network speeds

It can be seen that Y is faster than X and GNOME in nearly all instances. The main bottleneck is in starting an application. Currently, a round trip must be made for every widget that is instantiated. This, coupled with the inefficiency of the current socket implementation, makes start up times slower than they need be. A solution to this is presented in the future work section below.

10.3 Future Work

The intention of this project was to produce a foundation for the successor to X. As a result, there are many ways in which Y can be extended and improved. Some of these are listed below, but there will be many more.

Security

Y needs some form of security to allow only permitted users to make connections to the display. The security system should be easier to use than the current X authentication model, which confuses even expert users of X.

There are two possible ways to implement this. The server could create a certificate file when it starts or when the user logs in. When the user wants to use the same display from a different machine, he or she will copy the certificate file to an appropriate place on that machine. The client library and server will automatically negotiate certificates to try and find the appropriate one. Also, if the user wants to allow another user to access the server, he or she could send that user a copy of the certificate to use.

Alternatively, Y could maintain a persistent trust database, similar to that of SSH. When the server start up, it loads a key-ring from a file in the users home directory. When any application starts up, it must negotiate with the server in order to prove that it may access the server.

Additional Widgets

Y has only eight of the forty-something widgets that are present in most toolkits. Implementing these will be a priority, as many applications will depend on them.

Applications

Once the widget set is rich enough, work can begin writing or porting applications for Y. Further, cross-platform toolkits, such as wxWindows, Qt, or SWT can be implemented on Y to allow many existing applications to run.

Additional Client Libraries

In order to support as many languages as possible, appropriate class libraries for those languages can be written.

3D Rendering

An interface to OpenGL should be built which allows OpenGL applications to run on Y. This is probably best implemented as a new widget which exposes a GLCanvas. Initially software rendering should be implemented, though hardware acceleration should be possible using the DRI framework used by X and DirectFB.

Multimedia Playback

At the simplest level, multimedia playback can be achieved by providing a MediaCanvas widget, which exposes a YUV buffer. Colourspace transformation and scaling can then be implemented either in the server, or using hardware.

A better solution would be to provide a codec system within the server. Client applications could then send the video data to the server in its compressed form, and the server would automatically decode it and display it on a MediaCanvas. This approach should solve the current problem on GNU/Linux where there is no centralised repository of codecs. Because of this each application uses its own codecs which can result in different applications being able to play different types of file.

Further, a natural extension of the display server is to provide support for sound. The need for this is evidenced by the existence of KDE's MCOP protocol and the X consortium's media application server. By handling audio as well as video, the display server can synchronise the two correctly, as well as handle any contention for the limited number of output channels that some audio cards have.

Hardware Acceleration

Drivers for the many types of hardware will need to be written. Hardware that supports alpha blending data from memory should be exploited to improve the rendering speed. Hardware acceleration should also be tied in to 3D rendering and multimedia playback.

Internationalisation

Once Y has a coherent set of widgets, internationalisation of those widgets can begin. This can include support for extra keys on keyboards, complex input method editors that work on all widgets in the same way, support for right-to-left text, and perhaps even mirrored dialog boxes for right-to-left languages.

Automatic Configuration

Once proper device drivers are written, an automatic configuration system can be created. This would probe the hardware on initial startup and determine what is available. Recent graphics cards and LCD monitors also have the ability to suggest what the recommended resolution of the display is, and Y should take advantage of this.

Legacy X Protocol Handler

In order to support the wealth of X applications that already exist, and to ease the transition from X to Y, an interpretation layer will need to be built.

This is an excellent example of the elegance of the design of Y. The X layer can be implemented as an in-server driver module. This module would, upon initialisation, create an appropriate socket to pretend to act as an X server. When X applications connect to this socket, the X module would translate the requests into equivalent Y requests.

One drawback of supporting the X protocol is that many of the advantages of Y, in particular the lightweight protocol and server-side objects, will be lost.

Remote Desktop Server

In order to allow access to an already running Y server from another location on a network, it will be necessary to implement a remote desktop server within Y. This could be implemented as a video driver, as this will provide the framework that is needed.

For greatest portability, it would be best to implement a well known remote desktop protocol, such as that used by VNC [28]. This will allow a VNC client on any windowing system to connect to any Y server with this module loaded.

Atomic Interface Building

In order to solve the problem of start-up latency, the interface building operation needs to be made atomic, that is to make it possible to define an application's entire interface in one server round-trip.

A suggestion for this is to enable client applications to provide a high level description of their interface in something like XML. This description would contain details of how objects are packed into the layout widgets, plus default settings for the objects properties. An example XML fragment might look like:

```
<?xml version="1.0"?>
<y:window name="sample" id="0">
  <y:gridlayout id="1">
    <y:gridlayout:child attach="0,0" size="2,1">
      <y:label id="2" text="Testing" />
    </y:gridlayout:child>
    <y:gridlayout:child attach="0,1">
      <y:button id="3" text="OK" />
    </y:gridlayout:child>
    <y:gridlayout:child attach="1,1">
      <y:button id="4" text="Cancel" />
    </y:gridlayout:child>
  </y:gridlayout>
</y:window>
```

The Y server would respond with a mapping from the requested `id`s to the real `id`s of the created widgets.

An approach similar to this can be seen in DialoX [8].

Session Suspension and Transportation

Session suspension is the ability to serialise an applications state back to the client library, suspending the application, and restore it back to the server at a later date. Session Transportation is moving an application's server objects from one server to another.

Essentially these are same thing, as transportation is equivalent to suspending the session, and resuming it on a different server immediately.

This could be implemented by offering two new operations. The first would return a representation of all the clients objects to the client. The second takes this representation and restores the clients objects from them. Since the IDs may have changed, the server will need to return a mapping from old ID to new ID. If the serialised form is the same as the XML form described above, these two operations could be merged.

Appendix A

Source Code

A.1 Obtaining the Source

Y uses the Arch revision control system by Tom Lord. Users of Arch can obtain the latest revision by getting `Y--main` from:

```
mbt99@doc.ic.ac.uk--archive
    http://www.doc.ic.ac.uk/~mbt99/{arch}/
```

Arch is Free Software, and can be downloaded from

http://regexps.srparish.net/.

Alternatively, a recent distribution tar archive of Y can be found at

http://www.doc.ic.ac.uk/~mbt99/Y/.

A.2 Build Pre-requisites

Y is dependent upon:

- Linux kernel 2.4. Note that the radeonfb driver in the 2.4 series has a bug which affects some ATI Radeon cards.
- SDL 1.2.
- FreeType 2.1.3.
- SigC++ 1.0.
- Iterm 0.5. A patch to Iterm is needed to enable the home, end, page up, page down, insert and delete keys. The next version of Iterm will have these included.

A.3 Compiling and Running

Y uses the GNU Autotools, and so can be built using:

```
tar xfz Y-0.1.tar.gz
cd Y-0.1
mkdir build
cd build
../configure --prefix=/path/to/installation/directory
make
make install
```

Default configuration files will be installed to `etc/Y` in the installation directory. These may need to be modified. Note that Y must be installed prior to use as it must be able to find its plug-ins and configuration files.

To start the Y server, run `startY`. This will start the Y server and a terminal emulator. Further applications can be run from this terminal.

Appendix B

Terminology

B.1 Network Transparency

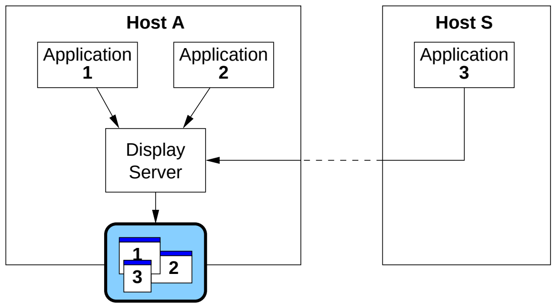


Figure B.1: Normal operation versus network transparency

In normal operation, applications connect to a display server that is on the same host as they are. There are many different communication primitives that applications can use in this situation, in particular shared memory.

An application can also be run on a remote host, and be adapted such that it makes its display connection to a display server running on another host. This is illustrated with Application 3 in figure B.1.

Network Transparency is when this is done automatically without any assistance from the programmer of the application, and with minimal assistance from the user (i.e. the user need only identify the display server they wish to connect to).

Of course, when operating over a network the communication primitives are severely limited, so things like shared memory must be emulated by the client library and server.

B.2 Remote Desktop Access

An entirely different situation is that of remote desktop access, as illustrated in figure B.2. Here, a client running on another machine (for example, a VNC [28]

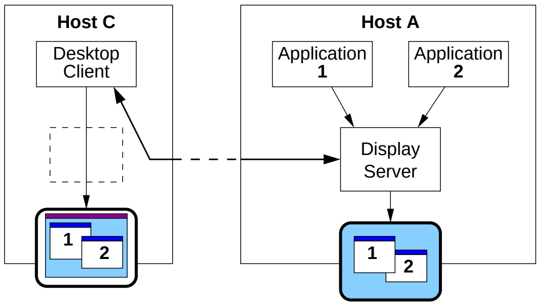


Figure B.2: Remote desktop access

client), makes a special connection to the display server on another machine, and replicates its display within a window on the client's host.

Usually updates are sent in the form of rectangles of bitmap data, so attention must be paid to compressing the data stream, which can end up being fairly large.

B.3 Objects, Widgets and Gadgets

The names for the different types of object within window systems vary from vendor to vendor. Y uses the following conventions, adapted from the popular X toolkits GTK and Qt.

Objects are server-side objects to which the client 'owns' and therefore has access to, and can encapsulate any data or behaviour that is necessary.

Widgets are the discrete graphical components on screen, such as buttons and labels. Generally these are also 'objects', so that the client may create and access them.

Gadgets are the sub-components of complex widgets. For example, a window might have a close gadget and a maximise gadget; a drop-down list box might have a drop-down button gadget. Generally, these will not be 'objects' and the client will manipulate them using members of the containing widget.

Appendix C

LibYc++ API Documentation

LibYc++ is the client library for writing Y applications in C++. The library provides a collection of classes which mirror the objects that are available on the server, allowing you to manipulate the server objects as though they were ordinary C++ objects.

LibYc++ uses libSigC++ to provide loosely coupled signals. Please read the libSigC++ documentation to find out more about how to use it.

The next few pages will give full documentation for each of the classes, along with some examples of how to use them.

Y::Y

This class is the main Y class, as it provides methods for creating and manipulating the connection to the Y server.

Members

static void initialise (int *argcp, char ***argvp)

Initialises the connection to the Y server. It will pull any pertinent options off of the command line, modifying `*argcp` and `*argvp` in the process.

static void run ()

Enters the main loop, transferring execution to the library.

static void time (int msec)

Sets a timer that will expire in no less than `msec` milliseconds. When the timer expires, the `timer` signal will be emitted.

static int findClassId (const char *className)

Queries the Y server to determine whether the class named `className` exists. This will return its class identifier if it does exists, or 0 if it does not.

Signals

static SigC::Signal0⟨void⟩ timer

This signal will be emitted when the timer set by `time` expires.

Example

This example shows the bare minimum that any Y application will want to do. It opens a window on the Y server, and sets its title.

```
#include <stdlib.h>
#include <Y/c++.h>

static void
closeApplication ()
{
  exit (EXIT_SUCCESS);
}

int
main (int argc, char **argv)
{
  Y::Y::initialise (&argc, &argv);

  Y::Window *window = new Y::Window ();
  window -> setProperty ("title", "Test Program");

  window -> requestClose.connect (SigC::slot (&closeApplication));

  window -> show ();

  Y::Y::run ();
  return EXIT_SUCCESS;
}
```

Y::Object

This class is the superclass of all server-based objects.

Members

static Object *find (int id)

Returns the object that has ID `id`, or `NULL` if that's not found.

int getId () const

Returns the ID of the object.

void setProperty (const string &name, const string &value)

Sets the property `name` to the string `value`.

void setProperty (const string &name, int value)

Sets the property `name` to the integer `value`.

string getProperty (const string &name)

Returns the value of the property `name` coerced to a string.

void subscribeSignal (const string &name)

Subscribes this application to the signal specified by `name`. From now on, this object will start getting events relating to this signal.

virtual void onEvent (const string &name, const vector⟨string⟩ &params)

This template method is called whenever a subscribed signal emission is received. The `name` is the name of the signal, and the `params` are the parameters associated with the signal.

Protected Members

void create (int classId)

Creates a server-side object with server class `classId` and associates this local object with the new object. This should be called by the appropriate constructor of the subclass.

vector⟨string⟩ invokeMethod (const string &nameAndParams, bool expectReturn = true)

This function invokes a remote method on the server object. `nameAndParams` is a pre-formatted string containing the method name and parameters separated by ASCII 'FS' characters (`0x1C`).

Example. To call a method ‘foo’ with parameters ‘1’, ‘2’, ‘3’, use:

```
#include <sstream>
...
  ostringstream ss;
  ss << "foo" << Y::Y::sep << 1 <<
                 Y::Y::sep << 2 <<
                 Y::Y::sep << 3;
  results = invokeMethod (ss.str ());
...
```

If `expectReturn` is true (the default), then the call is made synchronously to the server, and any results are returned in a vector of strings.

Y::Window

This represents an application's main window.

Constructor

Window ()

Creates a new Window. The window is not yet visible.

Properties

title

The title of the window.

Members

void show ()

Shows the window on the desktop.

void setChild (Y::Widget *widget)

Sets the window's contained widget to `widget`.

void setFocused (Y::Widget *widget)

Focuses the child widget `widget`.

Signals

SigC::Signal0⟨void⟩ requestClose

This signal is emitted when the user clicks the close button. The application should close unless there is good reason not to.

Y::GridLayout

This widget lays out its children in a homogeneous grid. The grid is automatically sized to contain all of its children.

Constructor

GridLayout ()

Creates an empty GridLayout.

Members

void addWidget (Widget *widget, int x, int y, int w=1, int h=1)

Adds a child widget. The child is placed with its top left portion in the cell in row `y`, column `x`, counting from (0, 0) at the top left. The child's width is set to `w` columns and `h` rows.

Y::Label

Constructor

Label ()

Creates an empty Label.

Properties

text

The text contents of the label.

alignment

The alignment of the text, one of `"left"`, `"center"` or `"right"`.

Y::Button

Constructor

Button ()

Creates an empty Button.

Properties

text

The text label of the button.

Signals

SigC::Signal0⟨void⟩ clicked

This signal is emitted whenever the button is clicked.

Y::CheckBox

The checkbox provides an on/off toggle switch.

Constructor

CheckBox ()

Creates an empty CheckBox.

Properties

text

The text label of the check box.

value

This is 1 if the check box is set, and 0 otherwise.

Signals

SigC::Signal1⟨void, bool⟩ clicked

This signal is emitted whenever the check box is clicked. The boolean contains the new value of the checkbox.

Y::Canvas

The canvas provides a drawable region of the screen. The canvas is double buffered, that means you draw to a "back" buffer, whilst a "front" buffer is being shown to the user. When you have finished drawing the back buffer, call `swapBuffers` to switch the two buffers and display what you have just drawn to the user.

Constructor

Canvas ()

Creates an empty Canvas.

Properties

background

Specifies the background colour in RGBA.

Signals

SigC::Signal0⟨void⟩ resize

This signal is emitted *once* when the canvas is resized. You must call `reset` in order to find out the new size. You will not receive any more resize notifications until you call `reset`.

Members

void savePainterState ()

Saves the painter's state (blend mode, pen colour and fill colour).

void restorePainterState ()

Restores the corresponding saved state.

void setPenColour (uint32_t)

Sets the pen colour to the RGBA colour specified.

void setFillColour (uint32_t)

Sets the fill colour to the RGBA colour specified.

Example. To set the pen colour to opaque red, and the fill colour to 50% transparent white, use:

```
canvas -> setPenColour (0xFF0000FF);
canvas -> setFillColour (0xFFFFFF80);
```

void reset (int &newWidth, int &newHeight)

Clears the canvas to its background colour and returns the size of the canvas in the newWidth and newHeight variables.

void clearRectangle (int x, int y, int w, int h)

Clears the rectangle specified to the fill colour.

void drawRectangle (int x, int y, int w, int h)

Draws the rectangle specified in the fill colour, with an outline in the pen colour.

void drawHLine (int x, int y, int dx)

Draws a horizontal line starting at (x, y) extending for `dx` pixels to the right in the pen colour.

void drawVLine (int x, int y, int dy)

Draws a vertical line starting at (x, y) extending for `dy` pixels down in the pen colour.

void drawLine (int x, int y, int dx, int dy)

Draws the line specified in the pen colour.

void drawHLines (int n, . . .)

Draws a series of `n` horizontal lines in the pen colour, specified by the groups of three parameters that follow.

void drawVLines (int n, . . .)

Draws a series of `n` vertical lines in the pen colour, specified by the groups of three parameters that follow.

void drawLines (int n, . . .)

Draws a series of `n` lines in the pen colour, specified by the groups of four parameters that follow.

Example. To draw a triangle in one operation, use:

```
canvas -> drawLines (3,   0,  0, 100,   0,
                        100,  0, -50,  75,
                         50, 75, -50, -75);
```

void swapBuffers ()

Swap the back and front buffers.

Bibliography

[1] Apple Computer, Inc. Aqua human interface guidelines. http://developer.apple.com/techpubs/.

[2] AtheOS. http://www.atheos.cx/.

[3] Michael Babcock. The importance of the GUI in cross platform development. In the Linux Journal, http://www.linuxjournal.com/article.php?sid=2723, 1998.

[4] Vannevar Bush. As we may think, 1945.

[5] David Canfield Smith. *Pygmalion, Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Massachusetts, 1993.

[6] Stuart J. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[7] Daniel Caujolle-Bert and Günter Barsch. On the xine-devel mailing list.

[8] DialoX. http://www.nar.fujitsulabs.com/dialox/.

[9] DirectFB. http://www.directfb.org/.

[10] The FreeType project. http://www.freetype.org/.

[11] The Fresco project. http://www.fresco.org/.

[12] The GNOME project. http://www.gnome.org/.

[13] Beverly L. Harrison, Hiroshi Ishii, Kim J. Vicente, and William A. S. Buxton. Transparent layered user interfaces: An evaluation of a display design. In *Proceedings of CHI'95: ACM Conference on Human Factors in Computing Systems*, pages 317–324, New York, 1995. ACM.

[14] Don Hopkins. *The UNIX Haters Handbook*, chapter 7, The X-Windows Disaster. IDG Books, Programmers Press, 1994.

[15] The KDE project. http://www.kde.org/.

[16] Brad A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2:64–103, March 1995.

[17] Keith Packard. Translucent windows in X.

[18] Ben Pfaff. GNU libavl. http://www.msu.edu/user/pfaffben/avl/.

[19] PicoGUI. http://www.picogui.org/.

[20] Thomas Porter and Tom Duff. Compositing digital images. *ACM Transactions on Computer Graphics*, 18:253–259, July 1984.

[21] Qt/Embedded. http://www.trolltech.com/products/embedded/.

[22] Jef Raskin. *The Humane Interface.* Addison Wesley and ACM Press, Reading, Massachusetts, 2000.

[23] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[24] Robert W. Scheifler, Jim Gettys, and Ron Newman. *X Window System C Library and Protocol Reference.* Digital Press, 1988.

[25] Ben Schneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, Massachusetts, 1998.

[26] Jiro Sekiba. Iterm.
http://oss.software.ibm.com/linux/projects/iterm/.

[27] Bjarne Stroustrup. *The C++ Programming Language (Third Edition)*. Addison Wesley, Reading, Massachusetts.

[28] VNC. http://www.uk.research.att.com/vnc.

[29] The XFree86 project. http://www.xfree86.org/.

[30] The XMMS project. http://www.xmms.org/.

[31] Shumin Zhai, William Buxton, and Paul Milgram. The partial-occlusion effect: Utilizing semitransparency in 3D human-computer interation. *ACM Transactions on Computer-Human Interaction*, 3:254–284, 1996.