X Window System Architecture Overview HOWTO

Daniel Manrique

roadmr@entropia.com.mx

Revision History

Revision 1.0.1	2001−05−22	Revised by: dm
Some grammatical corrections, pointed out by Bill Staehle		
Revision 1.0	2001−05−20	Revised by: dm
Initial LDP release.		

This document provides an overview of the X Window System's architecture, give a better understanding of its design, which components integrate with X and fit together to provide a working graphical environment and what choices are there regarding such components as window managers, toolkits and widget libraries, and desktop environments.

Table of Contents

1. Preface

This document aims to provide an overview of the X Window System's architecture, hoping to give people a better understanding of why it's designed the way it's designed, which components integrate with X and fit together to provide a working graphical environment and what choices are there regarding those components.

We explore several concepts that get mentioned a lot but might be a bit unclear for those without a technical background, such as widgets and toolkits, window managers and desktop environments. Some examples of how these components interact during day−to−day use of applications are provided.

This document is, deliberately, not too technically oriented. It's based on the author's (empirical) knowledge of the subject, and while it's primarily meant as a non−technical introduction, it can certainly benefit from any kind of comments, further examples and explanations, and technical corrections. The author welcomes all questions and comments regarding this document and can be reached at *roadmr@entropia.com.mx*.

2. Introduction

Back when UNIX was a new thing, around 1970, graphical user interfaces were only a weird thing being played with in a laboratory (Xerox's PARC to be precise). Nowadays, however, any operating system in hopes of being competitive needs to have a GUI subsystem. GUIs are supposed to be easier to use. This is not much of a concern under UNIX, which has traditionally been, to some extent, pretty user−hostile, preferring versatility over ease of use. However, there are several reasons why a GUI is desirable even on a UNIX system. For instance, given UNIX's multitasking nature, it's natural to have a lot of programs running at any given time. A GUI gives more control over how things are displayed on−screen, thus providing with better facilities for having a lot of programs on−screen at the same time. Also, some kinds of information are better displayed in graphical form (some, even, can only be displayed in graphical form; like pr0n and other inherently graphical data).

Historically, UNIX has had a lot of improvements done by academic types. A good example is the BSD networking code added to it in the late 1970's, which was, of course, the product of work at the University of California at Berkeley. As it turns out, the X Window System (also called X, but never X Windows), which is the foundation for most GUI subsystems found in modern UNIX (unices?), Linux and the BSD's included, was also the result of an academic project, namely the Athena project at the Massachusetts Institute of Technology (MIT).

Unix has been a multiuser, multitasking, timesharing operating system since its beginnings. Also, since the incorporation of networking technologies, it's had the ability to allow a user to connect remotely and perform work on the system. Previously this was accomplished either via dumb serial terminals, or network connections (the legendary telnet).

When the time came to develop a GUI system that could run primarily under Unix, these concepts were kept in mind and incorporated into the design. Actually, X has a pretty complex design, which has often been mentioned as a disadvantage. However, because of its design, it's also a really versatile system, and this will become quite clear as we explain how all the parts comprising a GUI under Unix fit together.

Before taking a look at X's architecture, a really brief tour of its history, and how it ended up on your Linux system, is in order.

X was developed by the Athena project, and released in 1984. In 1988 an entity called the "X Consortium" took over X, and to this day handles its development and distribution. The X specification is freely available, this was a smart move as it has made X almost ubiquitous. This is how XFree86 came to be. XFree86 is the implementation of X we use on our Linux computers. XFree86 also works on other operating systems, like the *BSD lineage, OS/2 and maybe others. Also, despite its name, XFree86 is also available for other CPU architectures.

3. The X Window System Architecture: overview

X was designed with a client−server architecture. The applications themselves are the clients; they communicate with the server and issue requests, also receiving information from the server.

The X server maintains exclusive control of the display and services requests from the clients. At this point, the advantages of using this model are pretty clear. Applications (clients) only need to know how to communicate with the server, and need not be concerned with the details of talking to the actual graphics display device. At the most basic level, a client tells the server stuff like "draw a line from here to here", or "render this string of text, using this font, at this position on−screen".

This would be no different from just using a graphics library to write our application. However the X model goes a step further. It doesn't constrain the client being in the same computer as the server. The protocol used to communicate between clients and server can work over a network, or actually, any "inter−process communication mechanism that provides a reliable octet stream". Of course, the preferred way to do this is by using the TCP/IP protocols. As we can see, the X model is really powerful; the classical example of this is running a processor−intensive application on a Cray computer, a database monitor on a Solaris server, an e−mail application on a small BSD mail server, and a visualization program on an SGI server, and then displaying all those on my Linux workstation's screen.

So far we've seen that the X server is the one handling the actual graphics display. Also, since it's the X server which runs on the physical, actual computer the user is working on, it's the X server's responsibility to perform all actual interactions with the user. This includes reading the mouse and keyboard. All this information is relayed to the client, which of course will have to react to it.

X provides a library, aptly called Xlib, which handles all low−level client−server communication tasks. It sounds obvious that, then, the client has to invoke functions contained within Xlib to get work done.

At this point everything seems to be working fine. We have a server in charge of visual output and data input, client applications, and a way for them to communicate between each other. In picturing a hypothetical interaction between a client and a server, the client could ask the server to assign a rectangular area on the screen. Being the client, I'm not concerned with where i'm being displayed on the screen. I just tell the server "give me an area X by Y pixels in size", and then call functions to perform actions like "draw a line from here to there", "tell me whether the user is moving the mouse in my screen area" and so on.

4. Window Managers

However, we never mentioned how the X server handles manipulation of the clients' on−screen display areas (called windows). It's obvious, to anyone who's ever used a GUI, that you need to have control over the "client windows". Typically you can move and arrange them; change size, maximize or minimize windows. How, then, does the X server handle these tasks? The answer is: it doesn't.

One of X's fundamental tenets is "we provide mechanism, but not policy". So, while the X server provides a way (mechanism) for window manipulation, it doesn't actually say how this manipulation behaves (policy).

All that mechanism/policy weird stuff basically boils down to this: it's another program's responsibility to manage the on−screen space. This program decides where to place windows, gives mechanisms for users to control the windows' appearance, position and size, and usually provides "decorations" like window titles, frames and buttons, that give us control over the windows themselves. This program, which manages windows, is called (guess!) a "window manager".

"The window manager in X is just another client −− it is not part of the X window system, although it enjoys special privileges −− and so there is no single window manager; instead, there are many, which support different ways for the user to interact with windows and different styles of window layout, decoration, and keyboard and colormap focus."

The X architecture provides ways for a window manager to perform all those actions on the windows; but it doesn't actually provide a window manager.

There are, of course, a lot of window managers, because since the window manager is an external component, it's (relatively) easy to write one according to your preferences, how you want windows to look, how you want them to behave, where do you want them to be, and so on. Some window managers are simplistic and ugly (twm); some are flashy and include everything but the kitchen sink (enlightenment); and everything in between; fvwm, amiwm, icewm, windowmaker, afterstep, sawfish, kwm, and countless others. There's a window manager for every taste.

A window manager is a "meta−client", whose most basic mission is to manage other clients. Most window managers provide a few additional facilities (and some provide a lot of them). However one piece of functionality that seems to be present in most window managers is a way to launch applications. Some of them provide a command box where you can type standard commands (which can then be used to launch client applications). Others have a nice application launching menu of some sort. This is not standardized, however; again, as X dictates no policy on how a client application should be launched, this functionality is to be implemented in client programs. While, typically, a window manager takes on this task (and each one does it differently), it's conceivable to have client applications whose sole mission is to launch other client applications; think a program launching pad. And of course, people have written large amounts of "program launching" applications.

5. Client Applications

Let's focus on the client programs for a moment. Imagine you wanted to write a client program from scratch, using only the facilities provided by X. You'd quickly find that Xlib is pretty spartan, and that doing things like putting buttons on screen, text, or nice controls (scrollbars, radio boxes) for the users, is terribly complicated.

Luckily, someone else went to the trouble of programming these controls and giving them to us in a usable form; a library. These controls are usually known as "widgets" and of course, the library is a "widget library". Then I just have to call a function from this library with some parameters and have a button on−screen. Examples of widgets include menus, buttons, radio buttons, scrollbars, and canvases.

A "canvas" is an interesting kind of widget, because it's basically a sub−area within the client where i can draw stuff. Understandably, since I shouldn't use Xlib directly, because that would interfere with the widget library, the library itself gives a way to draw arbitrary graphics within the canvas widget.

Since the widget library is the one actually drawing the elements on−screen, as well as interpreting user's actions into input, the library used is largely responsible for each client's aspect and behavior. From a developer's point of view, a widget library also has a certain API (set of functions), and that might define which widget library i'll want to use.

6. Widget Libraries or toolkits

The original widget library, developed for the Athena Project, is of course the Athena widget library, also known as Athena Widgets. It's very basic, very ugly, and the usage is not intuitive by today's standards (for instance, to move a scrollbar or slider control, you don't drag it; instead, you click the right button to scroll up and the left button to scroll down). As such, it's pretty much not used a lot these days.

Just as it happens with window managers, there are a lot of toolkits, with different design goals in mind. One of the earliest toolkits is the well−known Motif, which was part of the Open Software Foundation's Motif graphical environment, consisting of a window manager and a matching toolkit. The OSF's history is beyond the scope of this document. the Motif toolkit, being superior to the Athena widgets, became widely used in the 1980's and early 1990's.

These days, Motif is not a popular toolkit choice. It's not free (speech), and OSF Motif costs money if you want a developer license (i.e. to compile your own programs with it), altough it's OK to distribute a binary linked against Motif. Perhaps the best−known Motif application, for Linux users at least, is Netscape Navigator/Communicator (prior to Mozilla).

For a while Motif was the only decent toolkit available, and there's a lot of Motif software around. Of course people started developing alternatives, and there are plenty of toolkits, such as XForms, FLTK and a few others.

Motif is not heard of much these days, specially in the free software world. The reason is that there are now better alternatives, in terms of licensing, performance (Motif is widely regarded as quite a pig) and features.

One such toolkit, the widely known and used Gtk, was specifically created to replace Motif in the GIMP project (one possible meaning of Gtk is "GIMP ToolKit, altough, with its widespread use, it could be interpreted as the GNU ToolKit). Gtk is now very popular because it's relatively lightweight, feature−rich, extensible and totally free (speech). The 0.6 release of the GIMP included "Bloatif has been zorched" in the changelog. This sentence is a testament to Motif's bloatedness.

Another very popular toolkit these days is Qt. It was not too well−known until the advent of the KDE project, which utilizes Qt for all its GUI elements. We certainly won't get into Qt's licensing issues and the KDE/GNOME disjunctive. Gtk gets a lengthy mention because its history as a Motif replacement is interesting; Qt gets a brief mention because it's really popular.

Finally, another alternative worth mentioning is LessTif. The name is a pun on Motif, and LessTif aims to be a free, API−compatible replacement for Motif. It's not clear to what extent LessTif aims to be used in new development, rather than just helping those with Motif code use a free alternative while they (conceivably) port their apps to some other toolkit.

7. What we have so far

Up to this point we have an idea of how X has a client−server architecture, where the clients are our application programs. Under this client−server graphic system, we have several possible window managers, which manage our screen real estate; we also have our client applications, which are where we actually get our work done, and clients can be programmed using several possible different toolkits.

Here's where the mess begins. Each window manager has a different approach to managing the clients; the behavior and decorations are different from one to the next. Also, as defined by which toolkit each client uses, they can also look and behave differently from each other. Since there's nothing that says authors have to use the same toolkit for all their applications, it's perfectly possible for a user to be running, say, six different applications, each written using a different toolkit, and they all look and behave differently. This creates a mess because behavior between the apps is not consistent. If you've ever used a program written with the Athena widgets, you'll notice it's not too similar to something written using Gtk. And you'll also remember it's a mess using all these apps which look and feel so different. This basically negates the advantage of using a GUI environment in the first place.

On a more technical standpoint, using lots of different toolkits increases resource usage. Modern operating systems support the concept of dynamic shared libraries. This means that if I have two or three applications using Gtk, and I have a dynamic shared version of Gtk, then those two or three applications share the same copy of Gtk, both on the disk and in memory. This saves resources. On the other hand, if I have a Gtk application, a Qt application, something Athena−based, a Motif−based program such as Netscape, a program that uses FLTK and another using XForms, I'm now loading six different libraries in memory, one for each of the different toolkits. Keep in mind that all the toolkits provide basically the same functionality.

There are other problems here. The way of launching programs varies from one window manager to the next. Some have a nice menu for launching apps; others don't, and they expect us to open a command−launching box, or use a certain key combination, or even open an xterm and launch all your apps by invoking the commands. Again, there's no standarization here so it becomes a mess.

Finally, there are niceties we expect from a GUI environment which our scheme hasn't covered. Things like a configuration utility, or "control panel"; or a graphical file manager. Of course, these can be written as client apps. And, in typical free software fashion, there are hundreds of file managers, and hundreds of system configuration programs, which conceivably, further the mess of having to deal with a lot of disparate software components.

8. Desktop environments to the rescue

Here's where the concept of a desktop environment kicks in. The idea is that a desktop environment provides a set of facilities and guidelines aiming to standardizing all the stuff we mentioned so that the problems we mentioned earlier are minimized.

The concept of a desktop environment is something new to people coming for the first time to Linux because it's something that other operating systems (like Windows and the Mac OS) intrinsically have. For example, MacOS, which is one of the earliest graphical user interfaces, provides a very consistent look−and−feel during the entire computing session. For instance, the operating system provides a lot of the niceties we mentioned: it provides a default file manager (the finder), a systemwide control panel, and single toolkit that all applications have to use (so they all look the same). Application windows are managed by the system (strictly speaking there's a window manager working there). Finally, there are a set of guidelines that tell developers how their applications should behave, recommend control looks and placement, and suggest behaviors according to those of other applications on the system. All this is done in the sake of consistency and ease of use.

This begs the question, "why didn't the X developers do things that way in the first place?". It makes sense; after all, it would have avoided all the problems we mentioned earlier. The answer is that in designing X, its creators chose to make it as flexible as possible. Going back to the policy/mechanism paradigm, the MacOS provides mostly policies. Mechanisms are there, but they don't encourage people to play with those. As a result I lose versatility; if I don't like the way MacOS manages my windows, or the toolkit doesn't provide a function I need, I'm pretty much out of luck. This doesn't happen under X, altough as seen before, the price of flexibility is greater complexity.

Under Linux/Unix and X, it all comes down to agreeing on stuff and sticking to it. Let's take KDE for example. KDE includes a single window manager (kwm), which manages and controls the behavior of our windows. It recommends using a certain graphic toolkit (Qt), so that all KDE applications look the same, as far as their on−screen controls go. KDE further extends Qt by providing a set of environment−specific libraries (kdelibs) for performing common tasks like creating menus, "about" boxes, program toolbars, communicating between programs, printing, selecting files, and other things. These make the programmer's work easier and standardize the way these special features behave. KDE also provides a set of design and behavior guidelines to programmers, with the idea that, if everybody follows them, programs running under KDE will both look and behave very similarly. Finally, KDE provides, as part of the environment, a launcher panel (kpanel), a standard file manager (which is, at the time being, Konqueror), and a configuration utility (control panel) from which we can control many aspects of our computing environment, from settings like the desktop's background and the windows' titlebar color to hardware configurations.

The KDE panel is an equivalent to the MS Windows taskbar. It provides a central point from which to launch applications, and it also provides for small applications, called "applets", to be displayed within it. This gives functionality like the small, live clock most users can't live without.

9. Specific Desktop Environments

We used KDE as an example, but it's by no means the earliest desktop environment for Unix systems. Perhaps one of the earliest is CDE (Common Desktop Environment), another sibling of the OSF. As per the CDE FAQ: "The Common Desktop Environment is a standard desktop for UNIX, providing services to end−users, systems administrators, and application developers consistently across many platforms." The key here is consistency. However CDE wasn't as feature−rich and easy as it needed to be. Along with Motif, CDE has practically disappeared from the free software world, having been replaced by better alternatives.

Under Linux, the two most popular desktop environments are KDE and GNOME, but they're not the only ones. A quick internet search will reveal about half a dozen desktop environments: GNUStep, ROX, GTK+XFce, UDE, to name a few. They all provide the basic facilities we mentioned earlier. GNOME and KDE have had the most support, both from the community and the industry, so they're the most advanced ones, providing a large amount of services to users and applications.

We mentioned KDE and the components that provide specific services under that environment. As a good desktop environment, GNOME is somewhat similar in that. The most obvious difference is that GNOME doesn't mandate a particular window manager (the way KDE has kwm). The GNOME project has always tried to be window manager−agnostic, acknowledging that most users get really attached to their window managers, and forcing them to use something that manages windows differently would detract from their audience. Originally GNOME favored the Enlightenment window manager, and currently their preferred window manager is Sawfish, but the GNOME control panel has always had a window manager selector box.

Other than this, GNOME uses the Gtk toolkit, and provides a set of higher−level functions and facilities through the gnome−libs set of libraries. GNOME has its own set of programming guidelines in order to guarantee a consistent behavior between compliant applications; it provides a panel (called just "panel"), a file manager (gmc, altough it's probably going to be superseded by Nautilus), and a control panel (the gnome control center).

10. How it all fits together

Each user is free to choose whichever desktop environment feels the best. The end result is that, if you use an all−kde or all−gnome system, the look and feel of the environment is very consistent; and your applications all interact between them pretty nicely. This just wasn't possible when we had apps written in a hodgepodge of different toolkits. The range of facilities provided by modern desktop environments under Linux also enable some other niceties, like component architectures (KDE has Kparts and GNOME uses the Bonobo component framework), which allow you to do things like having a live spreadsheet or chart inside a word processing document; global printing facilities, similar to the printing contexts found in Windows; or scripting languages, which let more advanced users write programs to glue applications together and have them interact and cooperate in interesting ways.

Under the Unix concept of "desktop environment", you can have programs from one environment running in another. I could conceivably use Konqueror within GNOME, or Gnumeric under KDE. They're just programs, after all. Of course the whole idea of a desktop environment is consistency, so it makes sense to stick to apps that were designed for your particular environment; but if you're willing to cope with an app that looks "out of place" and doesn't interact with the rest of your environment, you are completely free to do so.

11. A day in the life of an X system

This is an example of how a typical GNOME session goes, under a modern desktop environment in a Linux system. It's very similar to how things work under other environments, assuming they work on top of X.

When a Linux system starts X, the X server comes up and initializes the graphic device, waiting for requests from clients. First a program called gnome−session starts, and sets up the working session. A session includes things such as applications I always open, their on−screen positions, and such. Next, the panel gets started. The panel appears at the bottom (usually) and it's sort of a dashboard for the windowing environment. It will let us launch programs, see which ones are running, and otherwise control the working environment. Next, the window manager comes up. Since we're using GNOME, it could be any of several different window managers, but in this case we'll assume we're running Sawfish. Finally, the file manager comes up (gmc or Nautilus). The file manager handles presentation of the desktop icons (the ones that appear directly on the desktop). At this point my GNOME environment is ready to work.

So far all of the programs that have been started are clients, connecting to the X server. In this case the X server happens to be in the same computer, but as we saw before, it need not be.

We'll now open an xterm to type some commands. When we click on the xterm icon, the panel spawns, or launches, the xterm application. It's another X client application, so it starts, connects to the X server and begins displaying its stuff. When the X server assigns screen space for my xterm, it lets the window manager (Sawfish) decorate the window with a nice titlebar, and decide where it will be on screen.

Let's do some browsing. We click on the Netscape icon on the panel, and up comes a browser. Keep in mind that this browser doesn't use GNOME's facilities, nor does it use the Gtk toolkit. It looks a bit out of place here... also, it doesn't interact very nicely with the rest of the environment. I'll open the "File" menu. Motif is providing the on−screen controls, so it's the Motif library's job to make the appropriate calls to the underlying Xlib, draw the necessary on−screen elements to display the menu and let me select the "exit" option, closing the application.

Now I open a Gnumeric spreadsheet and start doing some stuff. At some point I need to do some work on the xterm I had open, so I click on it. Sawfish sees that, and, being in charge of managing windows, brings the xterm to the top and gives it focus so I can work there.

After that, I go back to my spreadsheet, now that I'm finished I want to print my document. Gnumeric is a GNOME application, so it can use the facilities provided by the GNOME environment. When I print, Gnumeric calls the gnome−print library, which actually communicates with the printer and produces the hard copy I need.

12. Copyright and License