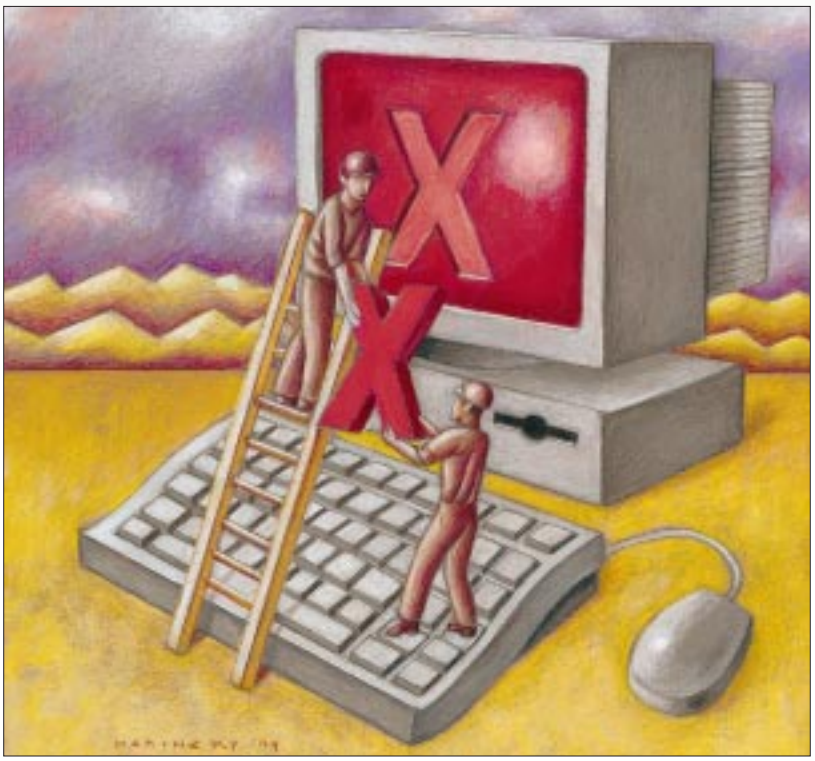
by Peter Collinson, Hillside Systems



JANE MARINSKY

Using X

Last month, I discussed the use of the `xterm` program. Several parts of the story were left dangling and I hope to fill in those holes with this month's column. As you probably know, `xterm` is a client for the X Window System, which has become the dominant method of providing GUIs on UNIX workstations.

The X Window System is an example of a "client/server" application; although many people were using X before the marketeers decided to use client/server as a buzzword. The hardware in your workstation–the bitmapped screen, keyboard and mouse–is controlled by an *X server*. *X clients* on your machine use the low-level X protocol to communicate with the X server, placing images on the screen. Input from your keyboard and mouse is captured by the X server, which directs data to the appropriate client.

To understand what's happening, let's start by thinking about a traditional UNIX application like a visual editor, `vi` or `emacs`, running on a nonwindowed system. Programs like `vi` and `emacs` execute commands directly by reading text from the keyboard: The text is interpreted and the stored file is changed. The editor reflects the changes in the file on the screen by sending the necessary hardware command sequences to the terminal, changing the image that the user sees. The programs are monolithic; they are large pieces of code that do the complete job of editing a file.

There is no reason why a program like `vi` should not be split into two sections: one that handles the interaction with the user and one that deals with editing the file. After all, these can be well-defined sections in the code. If we make the split and provide some communication between the two halves, then we've invented a new client that handles the user interface and a new server that handles the commands the user sends from the interface. The server takes in information from the client, edits the file and sends screen change commands back to the client.

Why would we want to do this? Well, Rob Pike (of Bell Laboratories) created his `Sam` editor in this fashion so that he could have a lightweight GUI running in an intelligent terminal. The GUI talks over a network to a more powerful system that runs the editor on the files stored on the remote machine. Rob was looking to make a clean split between the user interface and the main actions of the editor. Rob had other reasons too. Splitting the code into well-defined sections, which each take some input, perform an action and create some output, means the programmer is able to think more clearly about how each section of the program should be coded. The result is cleaner, more precise code.

Client/server working means we split the application into well-defined chunks, each undertaking some portion of the application. The chunks communicate using messages. At some point in the code of one chunk, the program will want to do something that uses a service provided by another chunk and

will create a message asking for the necessary action to be taken. The job of each chunk is to read a message, perform some action and send out a new message indicating that the action has been completed.

Once we have placed the requests for services into messages, there is no reason why we cannot pass those messages over a network, so that parts of the application can run on different machines. It all boils down to delivering the message to the appropriate server and making sure the result ends up at the correct place.

So I have created a picture where a single application is created from several constituent parts, all working together to achieve some work for the user. Often there are several applications where we want to provide multiplexed access to some shared resource. Coding such an application as a server accessed by a set of clients is a win because it's easier to control a single resource using a single program. Clients wishing to access the resource can do so by sending messages to the server, which can arbitrate between client demands in a controlled way. The server can take messages from several clients, each of whom are unaware of the others' existence. Messages from the clients can be sent in any order to be dealt with by the server, and a response is sent to the appropriate client.

There are two main ways to code such a server to cope with multiplexing. In the case of X, we are happy for the server to retain information about its clients and, in fact, we use the server to store data that is used by its clients.

Other applications have chosen to retain no state for their clients and simply act on the incoming messages in order of receipt. A Network File System (NFS) server is one example of this type of server. When NFS was designed, much was made of its "stateless" operation. Statelessness was designed in to ensure that when a client or server crashed and was rebooted, there was no need for complex recovery operations to resynchronize the client's view of its remote file system with the physical copy that was present on the server's machine. Both NFS and X servers handle requests from several clients controlling access to a shared set of resources, files for the NFS server and screen real estate for the X server.

The X Server

The X server is intended to be small and portable to many platforms. The aim is to provide a core server that supports only the functions that are needed to act as the glue for the windowing system. The X server handles the low-level output functions of creating and destroying windows, handling fonts and drawing text, lines, arcs, areas and bitmapped images. By the way, "windows" are more than just the area of screen real estate occupied by one client application. Each client may have several windows. For example, in the `xterm` program I used to create this document, there's a main text window and a scrollbar. In addition, `xterm` can create pop-up windows for menu dialogs and the like.

As I've mentioned before, the server accepts and processes keyboard and mouse events, sending them to the appropriate client. However, most of the work relating to input processing is done in the client, which allows it to have complete control.

The server manages several output resources for clients. It stores cursors and off-screen images called *pixmaps*. It looks after colormaps that control the mapping of colors onto the display. Many early display systems could only display 256 colors at any one time, but each one of these colors could be a full 24-bit color value. So an 8-bit color value is looked up in the color map to obtain the actual color that is to be displayed for any specific pixel on the screen.

The server also manages *graphical contexts*. There are many aspects of line and area drawing that can be configured and it's not efficient to send all the information each time a line is to be drawn. The client will set up a graphical context in the server and use it to dictate how several subsequent drawing actions are to be executed, thereby minimizing the amount of information that is sent from the client to the server for a particular set of drawing operations.

The server also manages a considerable amount of data for the clients, known as the *property database*. Some of this data is used for inter-client communication (such as cut and paste); and some is used to establish aspects of the look and feel and can be loaded from your `.Xresources` file.

Finally, the server manages the various network connections that are used to convey messages between itself and its clients. Most UNIX-based servers have several different methods that can be used to communicate with them. The server will either accept TCP connections from the network, establish a UNIX domain socket somewhere in the file system or use some form of shared memory connection.

There are several aspects of managing windows that you might expect to find in the server, but are off-loaded into clients. First, the server does not store the contents of the windows it manages. We might expect this because windows are dynamic objects that are covered by other windows, sometimes partially. When the user flips an obscured window on the screen from the back to the front, it's because the user wishes to see its contents. If the server stores the contents, it can redraw the window's contents and fulfil the user's request. However, X doesn't do this. Instead it's up to the client to redraw its windows (or some section of its windows) on the server's request. I'll guess this was done to save memory in the server.

Second, the server has no facilities for resizing windows. Resizing has to be done in the client, which then makes a request for screen real estate to accomodate its new window size and redraws the image.

Third, there is no support for any standard look and feel in the server. How each application should look is left completely up to the client. The client programmer can construct a look and feel that is thought to be desirable. This allows the programmer to create any look and feel they fancy, and permits a single platform to support a range of looks. Placing this utility into the client also makes development considerably easier, you don't need to take the X server up and down to create a new look.

Fourth, the server has no provision to handle "window manager" functions. A large part of the look and feel is provided by the window manager that supports the graphical infrastructure, allowing the user to manage their applications. Each display is

managed by a single window manager that intercedes between the clients and the server. The task of the window manager is to provide basic functions like virtual screens, resizing and movement of application windows, start/stop functions and swapping between large windows and icons.

Over the years, we've seen the rise and fall of many window managers–at present, I'm using two main ones. I use CDE on my Sun system and `fvwm` on my BSD/OS system. It's possible to switch trivially between window managers; you just kill one and start another. I notice that my newly booted Red Hat Linux system has a menu option that allows the user to switch between several managers, resulting in radical changes to the look and feel.

Using X Remotely

If you log in to a Sun workstation these days, then you will be using X, but you won't really notice the presence of the X server or any window manager. The server and the window manager will simply be there running for you. You only need to delve below the hood when you want to run X applications on other machines on your network. X allows you to run remote applications where their output appears on your screen and you can divert input from your keyboard and mouse into them.

I mentioned last month that I run `xterm` programs on various machines on my network, with their terminal emulator window sitting on my display taking input from my keyboard. My screen can be filled with several windows, each connected to a separate machine and each providing a command-line shell, allowing me to run commands on that machine. How do I do that?

Well, let's start at the beginning. One way is to start a local terminal window and use `rlogin` to access the remote machine. This works fine for terminal access, but isn't exactly what I want to achieve. I'd like to run the `xterm` program on the remote machine, but have its X events sent to the X server running on my workstation. Incidentally, we expect programs that have more graphical interfaces to run like this, and I certainly use `xterm` to test a new X setup, proving that the configuration is correct.

What happens if I log in to a remote machine and start the `xterm` program?

```
$ rlogin wooded
loads of login messages
I am now on wooded
$ xterm
Xterm Xt error: Can't open display:
$
```

I am greeted with an error message that hints something is missing. In fact, each X application looks in its environment for a variable called `DISPLAY`, which gives the name of the machine where the X server resides. Let's try again

```
$ rlogin wooded
loads of login messages
I am now on wooded
$ DISPLAY=craggy:0
$ export DISPLAY
$ xterm
```

Incidentally, if I was a `csh` user (or I was using one of its derivatives), I'd have said

```
% rlogin wooded
loads of login messages
I am now on wooded
% setenv DISPLAY craggy:0
% xterm
```

This time, I'm given another error message:

```
Xlib: connection to "craggy:0.0" refused by server
Xlib: Client is not authorized to connect to Server
xterm Xt error: Can't open display: craggy:0
```

However, something different is happening. The `xterm` program is actually trying to connect to my workstation, but my workstation is refusing the connection. Before pursuing this, let me explain a little more about the `DISPLAY` environment variable. The content of `DISPLAY` has the following general syntax: *machinename:displaynumber.screennumber*. The *machinename* entry is somewhat self-evident, it should contain the name of the machine that is your workstation. My network is arranged so that I can just use machine names without having to supply the fully qualified domain name. This may not be the case on your network.

The number that follows the colon (*displaynumber*) identifies a particular "display," where a display is a set of screens that share a common keyboard and pointing device. Most workstations only have one set, so the number is usually zero. However, the mechanism is there to provide support for large machines that have several distinct displays directly attached to them being used by several people.

The final number, the *screennumber*, is used to identify a specific screen in a multiple screen cluster. X can arrange to support more than one screen displaying applications for one user. Again, in most workstations that only have one keyboard, one mouse and one screen, this number is zero. When this value is zero, it can be omitted from the display specification, which I've done above.

Access Control

Well, back to the error message. Why is the connection being refused? In general, you don't want to permit just anyone to connect to the X server supporting your display. X doesn't provide any per-window access control; your window manager uses this lack of access control to manage the windows on the screen. In general, if someone can connect to your server, then they can access any of your windows and grab their contents. So it's important to be able to maintain control over who can and cannot access your X server.

You can throw all caution to the wind and tell your X

server to allow all connects from anywhere by using the `xhost` command:

```
$ xhost +
access control disabled,
     clients can connect from any host
```

This is useful when you are getting things going, but is not recommended practice for everyday use. However, to prove that things work, you can use this on your workstation and the `xterm` command on the remote machine will succeed. An `xterm` window should pop up on your screen and you will be able to type commands in from your workstation and have them run on a remote machine.

Having proved the connection will work, let's look at alternatives that offer better security. To reset access control, type

```
$ xhost -
access control enabled,
    only authorized clients can connect
```

One slightly better alternative is to use the access control list feature supported by the X server. With the default setting established to deny access to every other machine, we can use `xhost` to add a specific host to the list of machines that are permitted access:

```
$ xhost +wooded
wooded being added to access control list
```

We can now run the `xterm` command from `wooded` and only from that machine. Host-based controls might be sufficient in your environment, but if any other user can access the remote machine, then they can access your server, which might not be desirable.

The standard X system comes with a standard per-user authorization method. The idea is that the system stores a magic number (a magic cookie) in a secure file in your home directory (`.Xauthority`). When connecting to the server, the client reads the file and sends the cookie to the server. The server then permits access. The magic cookie is reset when you start the X server.

You may wonder exactly what help is provided by having a file that is local to the machine on which the X server is running. Well, if you share your home directory across all the machines in your environment, then things will work nicely. When you use `rlogin` to access the remote machine and start the `xterm`, then that `xterm` will find your `.Xauthority` file and send the cookie that will allow you access into the X server.

I am not able to do this on my network because I don't share my home directory across my machines. However, I usually leave my X server running for long periods and I have an authorization script that disseminates the cookie to the machines from which I plan to run remote terminals. The script uses the `xauth` program to manipulate the contents of the `.Xauthority` file on the remote machines.

Another alternative is to combine setting up the authorization with the actual setup of the `xterm`, creating a script that merges the necessary authorization information on the remote `.Xauthority` file and then calls a remote `xterm`. This would be something like

```
#!/bin/sh
local=`hostname`
# call this by xtc machinename
remote=$1
xauth nextract - "$local:0" |
     rsh $remote xauth nmerge -
rsh $remote \
     DISPLAY=$local:0.0 \
     exec xterm -sb -sl 1000\
     -T $remote -name $remote
```

The first line establishes a local variable that contains the name of the local machine using the backquote feature of the shell to run the `hostname` command and capture its result. Line 3 could bear some improvement; currently, it takes the name of the remote machine from the first argument to the script. It should check for the presence of an argument. Another possibility is to use the file name of the script to refer to the remote machine.

Having established these two constants, we use the `xauth` program to obtain the magic cookie from the local machine and merge it into the `.Xauthority` file on the remote machine. We do this by running `xauth` locally to extract the cookie in a numeric form (`nextract`), but writing it to the standard output. We run `xauth` remotely using `rsh` and merge the numeric value it finds by reading its standard input (`nmerge`) with the current contents of the `.Xauthority` file. Note the use of the common UNIX convention, where a hyphen on the command line means "use the standard input or output channels" instead of an explicit file name.

Having established the magic cookie in the remote `.Xauthority` file, we now use `rsh` to start the remote `xterm`. We ensure that the environment for the remote command contains the correct `DISPLAY` value and then use `exec` to replace the shell that `rsh` starts with the `xterm`. The arguments to `xterm` were discussed at length last month, so I'll say no more about them here.

As a final thought, it may be that you are running Kerberos on your site. This system can be used to establish rights for your clients to access your X server.

Further Reading

My personal touchstone when looking for a broad introduction to all things X is *The Joy of X, An overview of the X Window System* by Niall Mansfield (published by Addison-Wesley Publishing Co., 1993, ISBN 0-201-56512-9).

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests. He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.