Writing Your Own Clients [3]

This section describes the unique features of X programming on the Cray Research system, and it offers programming techniques to make the most efficient use of the Cray Research system. See the preface for a list of references that describe general X programming techniques.

The following topics are presented in this section:

- "Compiling a client" shows how to write a client.
- "Handling events," page 10, describes how to minimize network traffic, thus maximizing efficiency.
- "Using colors," page 11, discusses issues that pertain to color graphics.
- "Using fonts," page 14, describes techniques for efficient use of fonts.
- "Using images," page 15, discusses the use of client-side raster images.
- "Debugging tools," page 17, describes the use of the `cdbx`(1) and `xscope` debugging tools.

Compiling a client
3.1

When you build an X client on a Cray Research system, you must load the correct X libraries with the user-written code to create the executable binary file. The X11R5 libraries are as follows:

`libXext.a`	(Extension library)
`libXaw.a`	(Athena widget library)
`libXt.a`	(Intrinsics toolkit)
`libX11.a`	(also known as `Xlib`)
`libXmu.a`	(MIT utility library)

`libXau.a`	(sample authorization protocol for X)
`libXdmcp.a`	(X display manager control protocol library)
`libXi.a`	(`xinput` extension library)
`liboldX.a`	(X10 compatibility library)

Figure 1 shows the relationship of the Xlib, toolkit, widget, and extension libraries to each other.

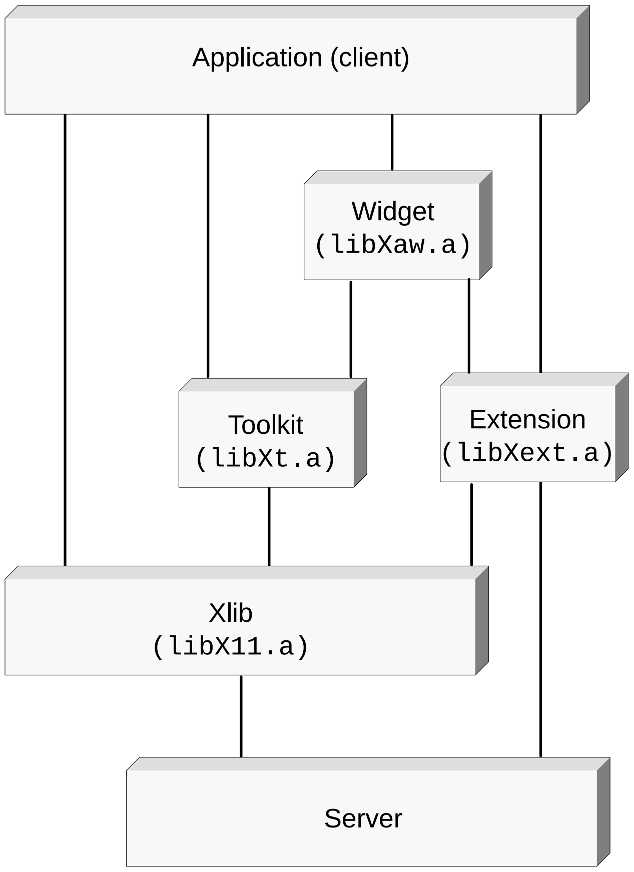

Figure 1. X11R5 libraries

You can use these same libraries when building a client on a workstation.

For a simple X client, you can use the following makefile:

```
# Set CFLAGS = –g for use with debugger
CFLAGS = –g  –DSYSV –DUSG
xclient: xclient.c
cc $(CFLAGS) –o $@ xclient.c \n
    –lXaw –lXt –lXmu –lX11 –lXext
```

Notice that SYSV and USG are defined. This is always a good practice when you are compiling applications that use X on the Cray system. If you are using an Imakefile, the define process is performed automatically. The imake(1) command takes an architecture-independent segment of a makefile and adds the architecture-dependent items to it. This provides an easy way to write a portable program.

If the resulting binary file is used with the debugger cdbx(1), you must use the –g option to cdbx to run the compile and load steps.

You can use the imake(1X) or xmkmf(1X) command to generate makefiles from templates known as *Imakefiles*. The following example shows a simple Imakefile:

```
DEPLIBS = $(DEPXLIB)
LOCAL_LIBRARIES = $(XLIB)
SimpleProgramTarget (test)
```

You can use either of the following commands to create a makefile from an Imakefile:

```
xmkmf

imake –DUseInstalled –DCURDIR=
```

Imakefiles released by Cray Research with UNICOS 8.0 have been upgraded to work with the Standard C preprocessor, usually located in /lib/cpp. Imakefiles obtained from other locations may still use the old format for comments and concatenation and therefore require the portable C preprocessor (pcpp). This preprocessor is located in /lib/pcpp on the UNICOS 8.0 operating system.

To designate a preprocessor other than the default (`/lib/cpp`) for either `imake` or `xmkmf`, set the `IMAKECPP` environment variable as follows:

```
setenv IMAKECPP /lib/pcpp        (C shell)

IMAKECPP=/lib/pcpp;              (Bourne shell,
export IMAKECPP                   Korn shell)
```

Handling events
3.2

The display server generates a packet of information for each occurrence of a user action, such as a keystroke, button press, mouse motion, window exposure, and so on, and each occurrence of interaction among programs. Each packet of information, called an *event*, is put into a protocol packet that consists of 32 bytes and is sent to each client that has requested that specific type of event. Each X protocol packet is also put into a TCP/IP packet, thus increasing the total number of bytes transferred. A client can request a specific type of event to track device input, another type of event to get mouse input, and so on. A client responds to events to control the user interface and to control communication among various clients. The processing that occurs when events are being controlled is known as *event handling*.

Keyboard input can generate a lot of network traffic. Each keystroke generates events `KeyPress` and `KeyRelease`, which typically cause a client to tell the server which character to display. The typing speed of individual users limits the network traffic, which is well below the network bandwidth. However, too many users typing into X clients on a Cray Research system can stress the communication resources.

Mouse motion is more stressful on network and CPU resources than keyboard input. Dragging the mouse across a window can generate numerous `MotionNotify` events. The server bundles together several `MotionNotify` events in one transmission to the client. The receiving client can compress `MotionNotify` events to appear as one event; however, because resources are wasted in generating, transmitting, and discarding them, a client should request `MotionNotify` events only when necessary. Alternatively, you can choose the

PointerMotionHints option, in which the server does the compression for you. Motion events are generated only under special circumstances, reducing the amount of data sent across the network.

The Xlib library provides the XSelectInput call, a mechanism that enables a client to choose the events it wants to see. One of the parameters of XSelectInput specifies the events that the server should generate for this client. Proper use of XSelectInput is the key to reducing unnecessary traffic between client and server. The following example shows the use of XSelectInput, in which keyboard input does not generate events, but mouse button activity does:

```
XSelectInput(display, window,
   ButtonPressMask | ButtonReleaseMask);
```

The following example shows a method for compressing mouse motion events in the client:

```
XEvent report;
while (XCheckTypedEvent(display, MotionNotify, &report));
```

XCheckTypedEvent reads the next MotionNotify event from the queue into the XEvent structure called report. If the event was a MotionNotify event, XCheckTypedEvent returns TRUE. The while loop discards all but the last MotionNotify event. This can be a great traffic reduction if the client has elected to see MotionNotify events.

Using colors
3.3

The visual impact of color allows you to convey a lot of information in a clear, concise manner. A color image can capture the large volume of numbers that a Cray Research system can generate and display it clearly. In the X Window System, colors can be used easily. However, if a client will be communicating with more than one server, you must take great care to ensure portability. The techniques described in this subsection can also greatly decrease the amount of network traffic that a client will generate when color is used.

Every X server has information about its particular color characteristics in a visual structure that you can obtain by using the DefaultVisual macro instruction or the XGetVisualInfo or XMatchVisualInfo functions. Several visual types are available; the most common are StaticGray (the typical visual for monochrome workstations) and `PseudoColor` (a common color visual in low-performance and medium-performance color workstations). Advanced color workstations may use other visuals, such as `TrueColor` or `DirectColor`.

To determine which visual is available for a client, you can use code similar to the following:

```
int my_visual;
Visual *visual;
visual = DefaultVisual(display, screen);
switch (visual–>class) {
     case StaticGray:
        my_visual = StaticGray;
        printf(”This is a StaticGray device\n”);
        break;
     case PseudoColor:
        my_visual = PseudoColor;
        printf(”This is a PseudoColor device\n”);
        break;
     case TrueColor:
        my_visual = TrueColor;
        printf(”This is a TrueColor device\n”);
        break;
     case DirectColor:
        my_visual = DirectColor;
        printf(”This is a DirectColor device\n”);
        break;
     default:
        printf(”ERROR in Visual.  Call programmer\n”);
        exit(1);
        break;
     }
```

To use colors in a client application, you can use the default colormap, or you can define a colormap, allocate one or more colors, and assign each color to a graphics context, which is then used for any drawing. Usually, you can use the server's default colormap (which is preferable), although for some applications, you might have to define a separate colormap; you should base your decision on whether the number of colors available in the default colormap is suffient for your application. A workstation with an 8-bit-per-pixel `PseudoColor` display is limited to 256 colors per colormap; a window manager might have allocated colors, other clients might have allocated colors, running xstdcmap might allocate colors, leaving a client with few colors to allocate.

You can either define colors by specific red, green, and blue intensities, or use any of several hundred predefined named colors. If you need only a few separate colors, it is much easier to use named colors to write a portable client program, although these colors may look somewhat different on various servers. The predefined named colors are defined in a dbm-format color name database on the server. It defaults to /usr/lib/x11/rgb. To display the color database, use the showrgb client or you may cat the database source file /usr/lib/x11/rgb.txt. The following example shows a color allocation and setting of a graphics context:

```
Colormap my_map;
XColor my_color;
GC my_gc;

     my_gc = XCreateGC(display, my_window, 0, 0);
     if(my_visual == PseudoColor) {
        my_map = DefaultColormap(display, screen);
        XParseColor(display, my_map, ”Red”, &my_color);
        XAllocColor(display, my_map, &my_color);
        XSetForeground(display, my_gc, my_color.pixel);
        }
     else {  /* use black if not PseudoColor device  */
        XSetForeground(display, my_gc, BlackPixel(display,screen));
        }
```

If the graphics application uses only the X Window System functions for line drawing, polygon drawing and filling, and text drawing, the use of color graphics does not greatly increase network usage. A simple drawing instruction that is sent from the client to the server is the same number of bytes, whether there is 1 bit per pixel, 8 bits per pixel, or 24 bits per pixel on the server screen. Thus, you can quickly draw on a window or a *pixmap* (a drawable mechanism that resides in the server memory) on either a black-and-white (monochrome) or a color workstation.

The additional overhead for allocating colors should be incurred only once during the client initialization. However, if you generate graphics on the client by using an `XImage` structure that resides in the client memory (see “Using images,” page 15), and then copy this image to the server, the network traffic is increased as the number of bit planes per pixel increases. Therefore, to minimize network traffic, an `XImage` structure should be avoided unless complex pixel-by-pixel manipulations are necessary.

Using fonts
3.4

Fonts are server resources that specify the style and size of print. Each server stores its own fonts. A remote client must be able to adapt to the fonts available on any given server. For example, because standard fonts and font names differ among X Window system versions, you must write clients to accommodate each potential server. A server can have both standard and nonstandard fonts installed. To be truly portable, however, a client should not depend on nonstandard fonts. To obtain a listing of the fonts available on a given server, use the `xlsfonts` command.

A client can use `XLoadQueryFont` to check the existence of a font before it tries to use that font. If you use `XLoadFont` with an unknown font name, an X protocol error occurs, but if you use `XLoadQueryFont` with an unknown font name, it does not. The following example uses `XLoadQueryFont`:

```
Display *d;
XFontStruct *theFont;

d = XOpenDisplay(argv[1]);

theFont = XLoadQueryFont(d,”bad_font_name”);
if (theFont == 0) {
    printf(”font name = %s, return = %d\n”, ”bad_font_name”,
theFont);
}
```

When using `XListFonts` in a remote client, you should be careful when using wildcard characters in a font name to match any font with a specific character string in it. For example, the string `*-times-medium-r-normal--*` matches the following font names:

```
-adobe-times-medium-r-normal--10-100-75-75-p-54-iso8859-1
-adobe-times-medium-r-normal--12-120-75-75-p-64-iso8859-1
-adobe-times-medium-r-normal--14-140-75-75-p-74-iso8859-1
-adobe-times-medium-r-normal--18-180-75-75-p-94-iso8859-1
-adobe-times-medium-r-normal--24-240-75-75-p-124-iso8859-1
-adobe-times-medium-r-normal--8-80-75-75-p-44-iso8859-1
```

However, the string `*-times-medium-r-normal--18-*` matches only the following font name:

```
-adobe-times-medium-r-normal--18-180-75-75-p-94-iso8859-1
```

If a client uses `XListFonts` to find all of the `*-times-medium-r-normal--*` fonts, and then examines each font by using `XLoadQueryFont`, a lot of traffic is generated on the network, because `XLoadQueryFont` returns information on every character in the font. Clients can experience several seconds delay by using `XListFonts` and `XLoadQueryFont` inefficiently.

Using images
3.5

An *image* is a client-resident representation of a screen area, not necessarily a window. Images differ from pixmaps or windows in that the image is created and stored on the client side; pixmaps and windows use server resources. The image, really a raster image, is a pixel representation in memory. You must take care when using images because various workstations interpret pixels differently, especially on color displays. (Even the definitions of black pixel and white pixel vary from manufacturer to manufacturer.) Some workstations use bit and byte ordering that is different from that used on other workstations. This further complicates matters.

Fortunately, Xlib provides all of the mechanisms necessary to convert images from client format to server format, so that an image can be sent to the server and displayed properly.

To overcome the problem of which pixel value represents black and which pixel value represents white, Xlib provides the `BlackPixel` and `WhitePixel` macros. In the following example, `XCreatePixmapFromBitmapData` requires values for the foreground and background pixel values. Using the `BlackPixel` and `WhitePixel` macros, you can ensure that the image will be correct, regardless of the server.

```
Display *dp;
Window winp;
Pixmap pixmapp;

dp = XtDisplay(toplevel);
winp = DefaultRootWindow(dp);
pixmapp = XCreatePixmapFromBitmapData(dp, winp, mandelbrot_bits,
       mandelbrot_width, mandelbrot_height,
       BlackPixel(dp, DefaultScreen(dp)),
       WhitePixel(dp, DefaultScreen(dp)), depth);
```

Overcoming the problem of byte and bit ordering requires that you modify the `XImage` structure, which is returned from the `XCreateImage` function call. The `byte_order` and `bitmap_bit_order` fields each contain a value that indicates most-significant bit first (MSBFirst) or least-significant bit first (LSBFirst) ordering. These refer to how programmers chose to represent the image in memory, not to the bit and byte ordering of the client or server hardware. Therefore, an image can be LSBFirst in the client and MSBFirst in the hardware.

You must know the contents of the XImage structure because the default values for byte_order and bitmap_bit_order are those of the server hardware, and the program must set these values to correspond to the actual byte order and bit order of the image. This can be done immediately after the XCreateImage call, as follows:

```
static XImage *xip = NULL;
Display *draw_d;
Visual *draw_v;
static char *dp = NULL;

dp=malloc (width * height);
xip = XCreateImage(draw_d, draw_v,
       depth, ZPixmap, 0, dp, width,
       height, bitmap_pad, byte_per_line);
xip->byte_order = MSBFirst;
xip->bitmap_bit_order = MSBFirst;
```

Debugging tools
3.6

One of the best tools available for debugging application programs on a Cray Research system is the cdbx(1) program. It has an X Window System interface, which displays the text being debugged, and it has command buttons for common operations such as run, stop at, step, next, and so on. If the DISPLAY environment variable is set to your display server, or the -display option is found on the cdbx command line, the X interface is enabled automatically. See cdbx(1) for details of its use.

Note

Reaching a breakpoint in cdbx during a server grab (a time when only the X program can use the server) hangs your display server. This happens only when cdbx and the X client are connected to the same server.

Another powerful tool for debugging X applications is `xscope`. This tool, which is in the public domain, traces all traffic between client and server. To the client, `xscope` appears as a separate display server because it uses a unique display number, then passes the protocol to the proper display.

A typical way of using `xscope` is to start it on your workstation with no parameters. This defaults to display 1 of the workstation. You must then set the `DISPLAY` environment variable to display 1 of the workstation and start the client.

The `xscope` tool intercepts each packet and displays it in an easy-to-read format. The following example shows this process. You can find an example of `xscope` output in "Example `xscope` Output," page 53.

Workstation	Cray system
`xscope`	–
–	`setenv DISPLAY mirror:1`
–	`xclient`
Protocol is traced here	–
`xclient` displays here	–

Sometimes it is difficult to determine the protocol request that actually caused a problem, because the client requests are buffered up; that is, several requests can be sent in the same transmission. To force Xlib to send requests in synchronous mode (as soon as they are built), you can issue the `XSynchronize` call from the `xclient` program. Using `XSynchronize` can make the `xscope` output easier to read, but this technique should be used only for debugging; unbuffered traffic increases network congestion. The following example shows the use of `XSynchronize`:

```
Display *display;
display = XOpenDisplay(argv[1]);
XSynchronize(display, 1);        /* force synchronous mode */
```

X toolkit clients can use the `–sync` command-line option to force synchronous mode, without editing or recompiling source code.