*Chapter 19*

# Using the GL in a Networked Environment

Network transparency is a built-in feature of the GL that allows a process on one IRIS workstation to display graphics either locally or over the network on a remote IRIS workstation.

## 19.1 Introduction

The network-transparent feature of the GL lets systems share the work load for graphics applications and lets servers without graphics capabilities use graphical tools.

For example, consider running a flight simulation to test a new aircraft design. You want to run a complex mechanical analysis with a simultaneous real-time animation. The mechanical analysis requires a "number-crunching" system and the animation requires a fast graphics display system. The two systems can share the work load, each doing the task for which it is best suited, in a client-server relationship, resulting in a more balanced work load and better overall performance.

The client / server model of the network-transparent GL allows remote display of graphics output. In the above example, a 4Server, acting as the client, performs the calculations for the mechanical analysis and sends the graphics calls over the network to an IRIS-4D workstation, acting as the graphics server, to display the flight animation.

### 19.1.1 Protocol

Network transparency is based on the Distributed GL (DGL) protocol that is built into the shared GL. The DGL protocol has two parts:

- a call mechanism built into the shared GL
- a graphics server to service requests made by DGL clients

In this chapter, the client application, which is linked with the shared GL, is called the *DGL client* and the graphics server is called the *DGL server*. In the DGL client, the DGL protocol sends tokens in a byte stream to the graphics server over the Ethernet® or other communication medium. The graphics server decodes the byte stream and calls the GL subroutines to display the graphics.

There is a separate product for running GL applications on non-IRIS hosts; see the documentation that comes with that option for more information.

### 19.1.2 Writing GL Programs to Use Network Transparent Features

Existing GL programs do not contain any calls that specifically invoke the DGL server. However, these programs can still be run remotely without modifying the source code, simply by relinking them with the shared GL (**-lgl_s**) and by linking with the Sun library (**-lsun**) if the Network Information Service (NIS) is desired.

Writing a network-transparent GL program is no different than writing a standalone GL program, except for optimizing performance.

Graphics calls are buffered from the client to the server, so you must flush the buffer periodically. The subroutine `gflush()` flushes the client buffer so GL calls can be received by the server.

#### gflush

The DGL client buffers calls to GL subroutines for efficient block transfer to the graphics server. The subroutine `gflush()` explicitly flushes the communication buffers and delivers all the untransmitted graphics data that is in the buffer to the graphics server.

GL subroutines that return data implicitly flush the communication buffers. In most programs, the implicit flushing that is performed by subroutines that return data is usually sufficient.

**Note:** All programs that are run over the network must call `gflush()` if the last command is a drawing command. No drawing is guaranteed to happen until `gflush()` is called.

The following situation illustrates a typical use of `gflush()`:

A program calls some Graphics Library subroutines that are buffered and not flushed. The program then either computes or blocks for a while, waiting for non-graphic I/O. `gflush()` must be called if the results of the buffered GL subroutines are to be seen on the host display before and during the pause.

Another reason for using `gflush()` is to reduce display jerkiness. If the client is computing data and then sending the data to the graphics server without implicit or explicit flushes, the data will arrive at the graphics server in large batches. The server may process this data very quickly and then wait for the next large batch of data. The rapid processing of GL subroutines followed by a pause results in an undesirable "jerky" appearance. In these cases it is probably best to call `gflush()` periodically. For example, a logical place to call `gflush()` is after every `swapbuffers()` call.

**Note:** Performing too many flushes can adversely effect performance.

### finish

`finish()` is useful when there are large network and pipeline delays. `finish()` blocks the client process until all previous subroutines execute. First, the communication buffers on the client machine are flushed. On the graphics server, all unsent subroutines are forced down the Geometry Pipeline to the bitplanes, then a final token is sent and the client process blocks until the token goes through the pipeline and an acknowledgment is sent to the graphics server and forwarded to the client process.

The following example illustrates a typical use of `finish()`:

A client calls GL subroutines to display an image. The subroutines all fit into the server's network buffers and the image takes 30 seconds to render. The client wants to wait until the image is completely displayed on the server's monitor before a message can be displayed on the client's terminal. `gflush()`

flushes the buffers, but does not wait for the server to process the buffers. `finish()` flushes the buffers and waits not only for the server to process all the graphics subroutines, but for the Geometry Pipeline to finish as well.

### 19.1.3 Establishing a Connection

To establish a connection, the client must have permission to connect to the graphics server. Permission is verified as it is for X clients. See the *xhost* man page for more information about client authentication procedures.

A server connection is established according to these rules:

1. If any of the following environment variables is defined, the server name is the value of the defined variable highest in the following list:
    1. DISPLAY
    2. DGLSERVER
    3. REMOTEHOST
2. If none of these environment variables are defined, then the server name is set to the client's hostname.

**Note:** The environment variables *DGLTYPE* and *DGLTSOCKET* are used for Silicon Graphics internal debugging purposes.

### 19.1.4 Using rlogin

If you use *rlogin* to log in remotely to an IRIS workstation, *REMOTEHOST* is defined. If *DGLSERVER* is undefined, the DGL protocol by default establishes a connection back to the last remote system where you ran *rlogin*. For example, if you *rlogin* from system A to system B and then *rlogin* from system B to system C, *REMOTEHOST* is set to B on system C. In this example the default graphics connection is B.

## 19.2 Limitations and Incompatibilities

The network-transparent GL has a few limitations and incompatibilities with the previous releases of the GL that was used strictly for local imaging. These limitations may prevent a GL application from executing properly only when remote connections are used.

### 19.2.1 The callfunc Routine

The `callfunc()` subroutine does not function in a GL program that is run remotely. Any references to `callfunc()` will result in a run-time error when executing the program.

### 19.2.2 Pop-up Menu Functions

A maximum of 16 unique callback functions are supported. Freeing pop-up menus does not free up callback functions. If you use too many callback functions, you get the client error:

```
dgl error (pup): too many callbacks
```

### 19.2.3 Interrupts and Jumps

You cannot interrupt the execution of a remotely called GL subroutine or pop-up menu callback function without returning back to that subroutine before calling another subroutine. This illegal condition typically results when you set an alarm or timer interrupt to go off and then block the program with a `qread()` call. If the signal handler does not return to the `qread()`, unpredictable results are likely (for example, it does a *longjmp*(3C) to some non-local location).

## 19.3 Using Multiple Server Connections

Connections to multiple graphics servers from one GL client program are supported and there are mechanisms for creating, multiplexing, and destroying server connections. You can use GL or mixed-model (X Window System and GL) subroutines for managing multiple server connections. Server processes normally reside on different server machines, but they can also reside on the same machine.

There are advantages to using multiple graphics servers, for example, some applications may require multiple windows, each with very high resolution graphics. Multiple windows on the same server must share one screen's resolution; however, with the network transparent feature of the GL, an application can control multiple servers, each of which can devote its full screen resolution to its windows.

Another possible application for multiple servers is improving performance when displaying multiple views of complex objects. If multiple views are displayed on multiple servers, performance is linearly increased by the number of servers. For example, an application can create a display list for a car on each of the servers that includes material and lighting parameters. Each server is given a different set of viewing parameters and then used to display the object.

A slight variation of the previous example is to have each server display a different representation of the object. For example, one server displays a depth-cued wireframe mesh of the car, another server displays a flat shaded polygonal representation of the car, and a third server displays a smooth shaded lighted surface representation of the car. If the display list for each of these representations is very large, multiple servers can eliminate or reduce paging, because each server needs only the display list for its representation.

### 19.3.1 Establishing and Closing Multiple Network Connections

The subroutines `dglopen()` and `dglclose()` allow a GL program to open and close graphics connections to server machines. You don't have to use these subroutines if your application is running on a single server because there is a default connection procedure, but you must use them if you are connecting to multiple servers.

### Using dglopen to Open a Connection

`dglopen()` opens a connection to a graphics server, and makes it the current connection. After a connection is established, all graphics preferences, input, and output are directed to that connection.

Communication remains enabled over the connection either until the connection is closed, or until a different connection is opened. A remote connection is closed by calling `dglclose()` with the *server identifier* returned by `dglopen()`. A different connection is selected by calling a subroutine that takes a graphics window identifier as an input parameter. The server connection associated with that graphics window identifier becomes the current connection.

To establish a connection, the client host must have permission to connect to the graphics server. Permission is verified as it is for X clients (see the *xhost* man page for more information about client authentication procedures).

To open a connection, you call `dglopen()` with a pointer to the server name (*svname*) and the type of connection you want.

Specify the server name as follows:

```
[[username]password@]hostname[:server[.screen]]
```

The *username* and *password* parameters are ignored; they are included for compatibility only. The *hostname* must be an Internet host name recognized by *gethostbyname*. If the connection succeeds, `dglopen()` returns the *server identifier*, a non-negative integer. Otherwise, `dglopen()` indicates a failure by returning a negative integer, the absolute value of which indicates the reason for failure.

Two types of connections are supported:

- DGLLOCAL is a direct connection to the local graphics hardware. This type of connection is not supported on client systems without IRIS graphics hardware.
- DGLTSOCKET is a TCP/IP socket connection to a remote host.

  Because you can mimic the behavior of a remote connection by using a DGLTSOCKET connection on a single machine, you can use the DGLTSOCKET connection during the development process to debug a remote application without connecting to another machine.

The following sequence of events occurs when a DGLTSOCKET connection is attempted:

1. The service *sgi-dgl* is looked up in */etc/services* to get a port number. If the service is not found, then an error occurs.
2. The server's name is looked up in */etc/hosts* or by the Network Information Service (NIS) to get an Internet address. If the host is not found, then an error occurs.
3. An Internet stream socket is created and some of its options are set.
4. A connection to the server machine is attempted with a small time-out. If the connection is refused, the timeout is doubled and the connection retried. If after several tries, the connection is still refused, an error occurs.
5. A successful connection is made and the server's Internet daemon invokes a copy of the graphics server. The graphics server process inherits the socket for communicating with the client program.
6. The graphics server uses the X authentication model to verify the login. Authentication is accomplished by the same mechanism as for X clients. See *xhost(1)* for more details.
7. The server process's group and user ID are changed according to the entry in */etc/passwd*.

### Using dglclose to Close a Connection

To destroy a graphics server process and its connection, call `dglclose()` with the *server identifier* returned by `dglopen()`. This terminates the graphics server process, freeing system resources, for example, open windows, that had been allocated and closes the graphics connection, freeing associated system resources on the client machine. Calling `dglclose()` with a negative *server identifier* closes all graphics server connections.

After `dglclose()`, there is no current connection. In order to resume rendering, you have to select another valid connection by calling a routine that takes a graphics window id as a parameter (such as winset) or you have to open another connection with `dglopen()`. Although it is not necessary, it is recommended that `dglclose(-1)` be called before exiting a GL application. This ensures that the graphics server processes exit cleanly.

### 19.3.2 Graphics Input

Each graphics server has its own keyboard, mouse, and optional dial and button box. The graphics input subroutines `qtest()`, `qread()`, `qdevice()`, `getvaluator()`, `setvaluator()`, and `noise()` execute on the current graphics server. The client program can therefore solicit input from multiple keyboards and mice. For most programs, it will make sense to get input from only one graphics server. In all cases, the programmer must make sure that the connection to the current graphics server is set correctly when graphics input is solicited.

### 19.3.3 Local Graphics Data

Each server process runs a separate copy of the GL and has its own local set of graphics data. For example, linestyles, patterns, fonts, materials, lights, and display list objects are local to each graphics server. When graphics data is defined, it is defined only on the current graphics server; other servers do not define it. You must be careful to reference local graphics data only on the server where it is defined. If a display list or font is used on multiple servers, it must be defined on each server.

### 19.3.4 Sample Program - Multiple Connections on a Local Host

This sample program illustrates how to establish multiple connections on a local host to solicit multiple graphics input.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#include <sys/types.h>
#include <sys/time.h>

static void DoLoop();

long main(int argc, char *argv[])
{
    int i;
    long wid1, wid2;
    fd_set readfds;
    long gl_fd1, gl_fd2;
    int nfound;
```

```
dglopen ("", DGLTSOCKET); /* force socket connection to local host */
wid1 = winopen("win 1");
qdevice(INPUTCHANGE);
qdevice(LEFTMOUSE);
qdevice(ESCKEY);
qdevice(REDRAW);

RGBmode();
gconfig();
cpack(0xff00ff);
clear();
finish();

dglopen ("", DGLTSOCKET); /* force socket connection to local host */
wid2 = winopen("win 2");
qdevice(INPUTCHANGE);
qdevice(LEFTMOUSE);
qdevice(ESCKEY);
qdevice(REDRAW);

RGBmode();
gconfig();
cpack(0x00ffff);
clear();
finish();

FD_ZERO(&readfds);

winset (wid1);
if ((gl_fd1 = qgetfd()) < 0) {
    printf("bad file descriptor %d\n", gl_fd1);
    exit(-1);
}

winset (wid2);
if ((gl_fd2 = qgetfd()) < 0) {
    printf("bad file descriptor %d\n", gl_fd2);
    exit(-1);
}

while(1) {
    FD_SET(gl_fd2, &readfds);
    FD_SET(gl_fd1, &readfds);
    nfound = select (getdtablesize(), &readfds, 0, 0, 0);
    printf("select nfound = %d\n", nfound);
```

```
        if (FD_ISSET(gl_fd1, &readfds)) {
             winset(wid1);
             DoLoop();
        }

        if (FD_ISSET(gl_fd2, &readfds)) {
             winset(wid2);
             DoLoop();
        }
    }
}

static void DoLoop()
{
    long dev;
    short val;

    while (qtest()) {
        dev = qread(&val);
        switch(dev) {
            case INPUTCHANGE:
                printf("INPUTCHANGE; wid = %d\n", val);
                break;
            case LEFTMOUSE:
                printf("LEFTMOUSE; val = %d\n", val);
                break;

            case REDRAW:
                printf("REDRAW; wid = %d\n", val);
                winset(val);
            clear();
                finish();
                break;

            case ESCKEY:
                gexit();
                exit(-1);

        }
    }
}
```

## 19.4 Configuration of the Network Transparent Interface

The DGL protocol software consists of two parts: a client library and a graphics server daemon. The client library is built into the shared GL (*/usr/lib/libgl_s.a*) and the graphics server daemon is */usr/etc/dgld*. The DGL protocol gets an Internet port number from */etc/services*, which has an entry for *sgi-dgl* (see *services*(4)).

### 19.4.1 inetd

The graphics server daemon for TCP socket connections is automatically started by *inetd*(1M). *inetd* reads its configuration file to determine which server programs correspond to which sockets. The standard configuration file, */usr/etc/inetd.conf*, has an entry for *sgi-dgl*. When a request for a connection is made:

1. The service *sgi-dgl* is looked up in */etc/services* to get a port number. If the service is not found, then an error occurs.
2. The server's name is looked up in */etc/hosts* or by the Network Information Service (NIS) to get an Internet address. If the host is not found, then an error occurs.
3. An Internet stream socket is created and some of its options are set.
4. A connection to the server machine is attempted with a small timeout allowance. If the connection is refused, the timeout is doubled and the connection retried. If after several tries, the connection is still refused, an error occurs.
5. A successful connection is made and the server's Internet daemon invokes a copy of the DGL graphics server. The graphics server process inherits the socket for communicating with the DGL client program.
6. The graphics server uses the X authentication model to verify the login. Authentication is accomplished by the same mechanism as for X clients (see *xhost(1)*).
7. The server process's group and user ID are changed according to the entry in */etc/passwd*.

### 19.4.2 dgld

The *dgld* daemon is the server for remote graphics clients. The server provides both a subprocess facility and a networked graphics facility. *dgld* is started by *inetd* when a remote request is received.

Local connections are not controlled by *dgld*; instead, a client program running on an IRIS host calls GL subroutines directly on the host machine. No authentication is performed for local connections.

TCP socket connections are serviced by the Internet server daemon *inetd*. *inetd* listens for connections on the port indicated in the *sgi-dgl* service specification. When a connection is found, *inetd* starts *dgld* as specified by the file */usr/etc/inetd.conf* and gives it the socket.

## 19.5 Error Messages

Error messages are output to a message file. The message file defaults to *stderr*. Error messages have the following format:

```
pgm-name error (routine-name): error-text
```

*pgm-name* is either *dgl* for client errors or *dgld* for server errors.

*routine-name* is the name of the system service or internal routine that failed or detected the error.

*error-text* is an explanation of the error.

### 19.5.1 Connection Errors

Table 19-1 lists the internally generated error values (defined in *<errno.h>)* that are reported when a connection fails.

| Error Value | Explanation |
|---|---|
| ENODEV | type is not a valid connection type |
| EACCESS | login incorrect or permission denied |
| EMFILE | too many graphics connections are currently open |
| ENOPROTOOPT | DGL service not found in */etc/services* |
| EPROTONOSUPPORT | DGL version mismatch |
| ERANGE | invalid or unrecognizable number representation |
| ESRCH | window manager is not running on the graphics server |

**Table 19-1** Error Values

### 19.5.2 Client Errors

Client error messages are output to *stderr*. For example, if NIS is not enabled and */etc/hosts* does not include an entry for the server host *foobar*, the following error message is output when a connection to is requested:

```
dgl error (gethostbyname): can't get name for foobar
```

If the client detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returns an error number (*errno* or *h_errno*), this number is used as the exit number.

Table 19-2 lists all exit values that are internally generated (not the result of a failed system call or service).

| Exit Value | Explanation |
|---|---|
| ENOMEM | out of memory |
| EIO | read or write error |

**Table 19-2** DGL Client Exit Values

The EIO value, accompanied by the message

```
dgl error (comm): read returned 0
```

usually means that communication with the server has been interrupted or was not successfully established. The configuration of the server machine should be checked (see Section 19.4).

### 19.5.3 Server Errors

Server error messages are output to *stderr* by default. For example, if */etc/hosts* does not include an entry for the client host, the following error messages are be output:

```
dgld error (gethostbyaddr): can't get name for 59000002
dgld error (comm_init): fatal error 1
```

The standard *inetd.conf* file runs the graphics server with the **I** and **M** options. The **I** option informs the graphics server that it was invoked from *inetd* and enables output of all error messages to the system log file maintained by *syslogd*(1M). The **M** option disables all message output to *stderr*.

If the DGL server is not working properly, check the system log file for error messages. Each entry in the *SYSLOG* file includes the date and time, identifies the program as *dgld*, and includes the process identification number (PID) for the server process. The rest of the error message is the text of the error message.

### 19.5.4 Exit Status

When the *dgld* graphics server exits, the exit status indicates the reason for the exit. A normal exit has an exit status of zero. A normal exit occurs when either the client calls `dglclose()` or when zero bytes are read from the graphics connection. The latter case can occur when the client program exits without calling `dglclose()` or terminates abnormally.

A non-zero exit status implies an abnormal exit. If the graphics server program detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returned an error number (*errno* or *h_errno*), this number is used as the exit number.

Table 19-3 lists all exit values that are internally generated (not the result of a failed system call or service).

| Exit Value | Explanation |
|---|---|
| 0 | normal exit |
| ENODEV | invalid communication connection type |
| ENOMEM | out of memory |
| EINVAL | invalid command line argument |
| ETIMEDOUT | connection timed out |
| EACCESS | login incorrect or permission denied |
| EIO | read or write error |
| ENOENT | invalid Graphics Library routine number |
| ENOPROTOOPT | dgl/tcp service not found in */etc/services* |
| ERANGE | invalid or unrecognizable number representation |

**Table 19-3** DGL Server Exit Value