1994

**DISplay User Manual Version 1.1**

Edward Mascarenhas

Vernon J. Rego  
*Purdue University, rego@cs.purdue.edu*

**Report Number:**  
94-082
DISplay User Manual
Version 1.1

Edward Mascarenhas and Vernon Rego

Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907
(317) 394-7835
{edm, rego}@cs.purdue.edu

Technical Report 94-082
## Contents

1 Introduction ................................................. 4

2 What is the DISplay Architecture? .................. 4
   2.1 Overview of what DISplay provides ................. 5

3 What does the User Interface look like? ........... 6
   3.1 The User Interface .................................. 6

4 Reaching Out to the Server ......................... 8
   4.1 sGetServer() .................................. 8
   4.2 sNewProcessMsg() ................................ 10
   4.3 sEndServer() .................................. 10

5 Graphical Displays ........................................ 11
   5.1 Task Setup and Termination ....................... 11
      5.1.1 sdCreateTask() ................................ 11
      5.1.2 sdSetTaskValues() .......................... 12
      5.1.3 sBeginTask() ................................ 12
      5.1.4 sEndTask() .................................. 12
   5.2 Plotting Line Graphs .............................. 15
      5.2.1 sdCreateSdPoints() .......................... 15
      5.2.2 sdAddSdPoint() ............................. 15
      5.2.3 sColPoint() ................................ 16
      5.2.4 sdFreeSdPoints() ........................... 16
      5.2.5 sPltPoint() and sPltPoint2D() .............. 16
      5.2.6 sRestartPlot() .............................. 17
   5.3 Histograms ........................................... 18
      5.3.1 sdFreqHistPoint() .......................... 18
      5.3.2 sHistPoint() ............................... 18
   5.4 Displaying the Level of a Variable .............. 19
      5.4.1 sDisplayLevel() ............................. 19
   5.5 Building a network ................................ 19
1 Introduction

DISplay is an application independent tool for performing visual interactive parallel simulations and computations. It provides the model developer with tools for parametric description of a large set of graphical outputs and capabilities for data visualization, inspection and specification. Abstract displays such as a plot of points and representational displays such as graphs are available as dynamic displays of the progress of a computation or as a means to display the end result. User interaction through dialogs allows for intervention in model execution. One way as well as two-way interaction between the user and the model is allowed.

2 What is the DISplay Architecture?

The architecture is based on the client-server paradigm\(^1\). The simulation application itself is treated as a client and is created by a model developer or programmer. Client calls are made by calling functions from the DISplay client library. The server part of the software consists of a connection server (CS) to which clients connect and a DISplay Server (DS) that handles all graphics requests and user interaction. Typically, at initialization the client connects to the CS which "forks" off the DISplay server and hands out to the application a handle (socket) to the server. The DS reads incoming messages from the application and creates the necessary display "tasks" and interaction "dialogs" as specified by the application. Continuing display and user interaction is based on the specifics and objectives of the application. The DS connects to an X server to carry out the necessary display and user interaction commands. The simulation analyst or user interacts with the application from the workstation where the display is being made. The architecture is shown in Figure 1.

The CS performs an important function in the above mechanism. It must be located at a well known machine and port so that applications can connect to it. Also applications connect to it using a unique "Name" which is used to identify the application. The connection server maintains a database of all DS's currently active and their corresponding "names". In a parallel computation environment there may be several processes (running for example on a cluster of workstations) which may want to cooperate to form graphical outputs and user interactions. In this case the CS hands out to all these processes (from the same application), the same handle so that all of them connect to the same DS. In this configuration the processes may run on remote machines, the CS and the DS may also run on machines that are not on the users desktop where the display is being done. Once the DS has been created, the application only interacts with the DS. The CS continues to

---

\(^1\)Users need not be specifically aware of this to be able to use DISplay
read connection requests from other applications. If the name specified by the incoming connection request is different from any of the current ones in the CS database then it will handout a connection to a new DS, otherwise, it hands out an existing connection. This is the primary means for sharing the server among several cooperating processes, and is important for parallel computing environments.

2.1 Overview of what DISplay provides

A fundamental design decision in the development of DISplay server was “application independence”. What this means is that it would not store any application specific state. This is what makes the DS general and allows it to be used by a wide range of applications. The DS does store information from the application for display purposes, but it is only the simulation analyst that makes decisions based on this and interacts with the application through the server. The DS only does the functions of presenting information to the user in a well defined manner, and accepting input, which is passed on to applications that are designed to accept this input. The interpretation of the presentation is left to the user. Functions that the DS provides can be carried out independent of the application. For example, clicking on a push button to display a graph would not affect the application in any way. Since the DS is X Windows based it provides the familiar windows, icons, menus, and pointing devices interface. The DS basically provides a point and click interface, to invoke functions at the DS, which can be carried out in random order if they do not interact with the application.

The DISplay software provides “tasks” and “dialogs” as the basic means of implementing graphical displays and user interaction respectively. DISplay tasks are of two types. “Single message” tasks require a single message to be delivered and their action is taken based on
that message alone. "Multiple message" tasks have a setup phase where the task is created locally, and then begun at the server. Creation returns a "task identifier" which is used as a handle in all subsequent messages related to the task. Task creation may be followed by sending task related messages to the DS which are interpreted in the context of the task identifier. Tasks are deleted by using a "end" task message. The use of a task identifier provides for some useful performance improvements. Thus it is not necessary to repeat whether a plot is 2 dimensional or 3 dimensional with every message that refers to this plot. This information is stored at the Server at creation time, and a request for plotting a point is executed based on whether the plot is setup as 2D or 3D. User interactions are also handled in a similar manner. Each possible user-model interaction is assigned a unique "dialog identifier" and a "form" based dialog is created. A dialog setup phase creates the dialog locally and then sets it up at the DS. Queries and answers using these dialogs are interpreted based on the dialog identifier and the position of the question within a dialog. Messages that are used for setup and termination are identified as "control" messages and other are referred to as "action" messages within the system. It is also possible to link dialogs to certain tasks and to have tasks linked to dialogs. For example the node or arc of a graph task can be associated with dialogs. When a dialog is activated it may simply display an active task like a plot of points.

3 What does the User Interface look like?

3.1 The User Interface

The DISplay Server provides the user (who is the simulation analyst in this case) with an user interface that can be modified depending on the nature of the application. A sample screen dump of an Performance Monitoring display created by DISplay is shown in Figure 2. The figure shows an example of a network (GRAPHTASK), a histogram task and several dialogs. The interface consists of the "Interaction" push-buttons as shown in the figure. "Panels" pop up when any of the interaction push-buttons are clicked on. Each panel encapsulates a group of associated dialogs and consists of a variable number of buttons, each of which invokes a dialog or pops up a display. The function of each button is displayed as its label. This graphical interface is common to all applications using the DISplay software. Each interaction push-button displayed on the main menu bar of the user interface is very application specific. To modify the presentation to the user (for example, to add another option in a panel) it is necessary to incorporate the option in the program code and recompile the user application. Thus creating application specific interfaces does not require the client library or the server software to be modified. Task windows are opened up whenever the Task is begun in the application. The X window system allows
Figure 2: The DISplay User Interface
the user to arrange multiple windows on the screen as preferred. The DISplay server also
handles redraw and resize commands independent of the application. Scale sizes and limits
can also be interactively changed and the complete graph is redrawn just as if it had started
with the new sizes. Windows can be iconified and deiconified as necessary. Dialogs can be
brought up to display data and then closed when not required.

The software may be used online as well as off-line. A typical online use proceeds as
follows. On Unix based machines the connection server can be started using the command
\texttt{condsrv [port server-port]}. The connection server accepts connections from \texttt{port}
and hands out server connections to clients starting from \texttt{server-port}. Default values for \texttt{port}
and \texttt{server-port} are 4000 and 4500 respectively. To bring up the display on a machine
other than the one where the DISplay server runs it is necessary to append the simulation
name with the name of the machine where the X server is running. For example if the
server is running on “ariadne.cs.purdue.edu” to obtain the display for the “M/M/n” server
application on “carcassi” the name could be provided as “M/M/n-display carcassi:0”. For
this reason it is best if the application is written to read in the simulation name from a file
or accept it from the terminal.

To run the software off-line the logging feature must be turned on from the \texttt{sNewProcessMsg()}
function call during an actual run and the display of windows may be turned off. Later the
logged messages may be input to the \texttt{dissdlog} program. This program accepts a command
line argument which is the file name of the log file. The program reads the logged messages,
and behaves as if the actual run were taking place. All synchronization is done exactly as
in the actual run. This feature may be useful as a demonstration tool when the dynamics
of some application has to be described to others.

\textbf{Note:} All client functions that return an integer, return the value SDOK or SDERROR
unless otherwise mentioned.

4 Reaching Out to the Server

4.1 \texttt{sGetServer()}

A connection to the server must be made before any display or interaction can be done.
This may be accomplished with the call:

\begin{verbatim}
int sGetServer(char *name, char *host, char *port)
\end{verbatim}

\begin{itemize}
  \item \texttt{name}: Unique name for the application, with optional display location.
\end{itemize}
main(int argc, char **argv)
{
    char *simname; /* the name of the simulation */
    char *host;    /* the location of the connection server */
    char *service; /* the port number of the connection server */
    int sock;     /* the handle to the DS */
    int ret;      /* return code from functions */
    int window=TRUE, log=TRUE; /* window up and logging on */
    int num_of_procs; /* number of processes in the simulation */

    ...;

    sock = sGetServer(simname, host, service); /* make the connection */
    if (sock == SDERROR) {
        /* handle the error here */
    }
    /* register this process as one of num_of_procs */
    ret = sNewProcessMsg(sock, window, log, num_of_procs);

    /* rest of processing and messages to DS */
    ret = sEndServer(sock); /* close the connection */
}

Figure 3: Example of connecting to the Display Server
• host: The domain name of the machine that runs the connection server.
• port: The port number on which the connection server runs.

4.2 sNewProcessMsg()

A call to connect to the server is followed by a call to identify oneself as a separate process using the call:

```c
int sNewProcessMsg(int sock, int window, int logging, int num_procs)
```

• sock: This is the handle to the server returned by a prior sGetServer() call.
• window: A value of TRUE indicates that a window will be created for this process. If it is false the window will not appear on the screen.
• logging: Currently this may be set as TRUE or FALSE depending on when logging is desired.
• num_procs: The number of processes that are connecting to the server and are part of the same application. By giving a value greater than 1 it is ensured that all processes connect before beginning to display data from the application.

4.3 sEndServer()

The last call to the DISplay server is to close the connection. Any calls from the DISplay client library may fail after the connection is closed. This is accomplished by making the call:

```c
int sEndServer(int sock)
```

• sock: This is the handle to the server returned by a prior sGetServer() call.

Figure 3 shows how the above functions may be used.
### Table 1: Values for Task Types in the DISplay client calls

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLOTTASK</td>
<td>Plot of points without ruler scales</td>
</tr>
<tr>
<td>HISTTASK</td>
<td>Histogram task</td>
</tr>
<tr>
<td>LEVELTASK</td>
<td>Level/meter</td>
</tr>
<tr>
<td>GPLOTTASK</td>
<td>Plot of points with X and Y scales</td>
</tr>
<tr>
<td>PLOT3DTASK</td>
<td>A three dimensional plot of points</td>
</tr>
<tr>
<td>GRAPHTASK</td>
<td>Network graph</td>
</tr>
</tbody>
</table>

#### 5 Graphical Displays

DISplay provides two types of displays which may be termed as abstract and representa­tional displays. Abstract displays show relevant model data in some alternate comprehensible form. For example a plot of a variable against time is an example of an abstract display. Representational displays shows the system under study in some simplified form from that of its real appearance. A network graph could be an example of a representational display. In DISplay all displays are knows as Tasks. Below is given a list of various types of display tasks that are available and their setup and use is explained.

#### 5.1 Task Setup and Termination

##### 5.1.1 sdCreateTask()

Tasks in DISplay are created by using the task control function:

```c
unsigned char sdCreateTask(int type, int window, int logging, int global)
```

The return value of the call is a unique task identifier (tid) which will be used in all subsequent calls pertaining to this task.

- **type**: This is a constant defined by the DISplay system. Use a type corresponding to the task being created from table 1.

- **window**: A value of TRUE indicates that a window will be created for this task. If it is false the window for this task will not appear on the screen.
• **logging**: Currently this may be set only as TRUE. Logging is always done for the task if it is being done for the process that created this task.

• **global**: Indicates whether this task is global or not. If it set to TRUE then all processes that create the task will get the same identifier and will be able to draw to the common task. In any application all global tasks must be created first and are shared by all the processes (so all processes must create them). Non shared tasks may be created after global tasks have been created.

### 5.1.2 sdSetTaskValues()

Creation of the task is followed by setting the parameters for the task. This is accomplished by the call:

```c
void sdSetTaskValues(unsigned char tid, ...)
```

The values of various parameters are specified in the call in pairs, with a DISplay name for the parameter and the value for the parameter. Any number of these may be specified in the same call. The list is completed by a NULL (see example in Figure 4). The various DISplay names for parameters are given in Table 2.

### 5.1.3 sBeginTask()

A task is begun at the server using the call:

```c
int sBeginTask(int sock, double time, unsigned char tid)
```

- **sock**: This is the handle to the server returned by a prior sGetServer() call.
- **time**: The simulation time at which this task is begun at the Server.
- **tid**: The task identifier returned by a prior sdCreateTask() call.

### 5.1.4 sEndTask()

Tasks end when the following call is made. The screen will still retain the task, however no more messages may be sent to the server pertaining this task.

```c
int sEndTask(int sock, double time, unsigned char tid)
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SdXaxisName</td>
<td>String</td>
<td>The name for the X axis</td>
</tr>
<tr>
<td>SdYaxisName</td>
<td>String</td>
<td>The name for the Y axis</td>
</tr>
<tr>
<td>SdZaxisName</td>
<td>String</td>
<td>The name for the Z axis</td>
</tr>
<tr>
<td>SdColor</td>
<td>String</td>
<td>Color name</td>
</tr>
<tr>
<td>SdXmin</td>
<td>double</td>
<td>Minimum value for X axis</td>
</tr>
<tr>
<td>SdXmax</td>
<td>double</td>
<td>Maximum value for X axis</td>
</tr>
<tr>
<td>SdYmin</td>
<td>double</td>
<td>Minimum value for Y axis</td>
</tr>
<tr>
<td>SdYmax</td>
<td>double</td>
<td>Maximum value for Y axis</td>
</tr>
<tr>
<td>SdZmin</td>
<td>double</td>
<td>Minimum value for Z axis</td>
</tr>
<tr>
<td>SdZmax</td>
<td>double</td>
<td>Maximum value for Z axis</td>
</tr>
<tr>
<td>SdNoSeries</td>
<td>Integer</td>
<td>Number of series of points in Histogram/Plot</td>
</tr>
<tr>
<td>SdDim</td>
<td>Integer</td>
<td>2D=2, 3D=3</td>
</tr>
<tr>
<td>SdTitle</td>
<td>String</td>
<td>Title for the Window</td>
</tr>
<tr>
<td>SdXaxisInterval</td>
<td>Integer</td>
<td>The X axis interval for an Histogram</td>
</tr>
<tr>
<td>SdEditable</td>
<td>Integer</td>
<td>Is the graph Edit-able, currently FALSE</td>
</tr>
<tr>
<td>SdOrientation</td>
<td>Integer</td>
<td>This is either HORIZONTAL or VERTICAL</td>
</tr>
<tr>
<td>SdMaxNodes</td>
<td>Integer</td>
<td>Maximum number of nodes in the graph</td>
</tr>
<tr>
<td>SdMaxArcs</td>
<td>Integer</td>
<td>Maximum number of arcs in the graph</td>
</tr>
</tbody>
</table>

Table 2: Values for Parameters in sdSetTaskValues client calls
```c
#include "sdconsts.h"
#include "sdclnt.h"
#include "sdutil.h"
{
    /* dialog */
    static SdQtnAns qtnans_sync[] = {
        {"What is your name?", SDSTRING},
        {"What is your key?", SDINT},
    };
    char *reply[2];
    double timereply=0.0;
    int i;

    int histtask;
    int dialog_task, dialog_sync;
    int global = 1;  /* is a global task */

    /* create and initiate a Histogram task */
    histtask = sdCreateTask(HISTTASK, window, log, global);
    sdSetTaskValues(histtask,
        SdXaxisName, "X- axis",
        SdYaxisName, "Y- axis",
        SdTItle, "Particle simulation",
        SdXmin, 0.0,
        SdXmax, 20.0
        SdYmin, 0.0,
        SdYmax, 20.0
        SdNoPoints, 5.0,
        SdDim, 2,
        SdXaxisInterval, 1,
        NULL);
    sBeginTask(sock, simtime, histtask);

    /* create the task dialog */
    dialog_task = sdCreateDialogTask("Histogram", histtask);
    /* create the Synchronous dialog */
    dialog_sync = sdCreateDialogNotask("PassKey", SDMODEL_QUERY,
        Number(qtnans_sync), qtnans_sync,
        NULL);

    /* begin the dialogs grouping the two into a single pane*/
    sBeginDialog(sock, simtime, dialog_task, dialog_sync, "Some Dialogs");

    /* invoke the synchronous dialog */
    ret = sQueryReply(sock, &timereply, dialog_sync, reply);

    /* print the replies */
    for (i = 0; i < Number(qtnans_sync); i++) {
        printf("Qtn:%s Ans:%s\n", qtnans_sync[i].question, reply[i]);
    }
}
```

Figure 4: Use of Task and Dialog Functions
5.2 Plotting Line Graphs

DISplay provides a simple means to plot lines between points. Several plots may be made on the same window so that it is easy to compare among various alternatives that the plot lines may represent. The relevant types (to be specified in the sdCreatetask() call) for 2 dimensional plots are PLOTTASK, and GPLOTTASK and for a 3 dimensional plot it is PLOT3DTASK. Axis related parameters must be specified for plot point tasks by using a sdSetTaskValues() function that was explained above. But first here’s how points on the window can be colored.

The following functions may be used to color a point on a “plot points type” of a task window with any color that is supported. Usually a bunch of points are displayed at the same time. A collection of points is stored in a SdPoints structure.

5.2.1 sdCreateSdpoints()

A SdPoints collection is created by the call:

SdPoints *sdCreatePoints(int count)

- count: The number of points that is to be created.

5.2.2 sdAddSdPoint()

Each point in the window that must be colored can be be added to the SdPoints structure using the call:

int sdAddSdPoint(SdPoints *points, double x, double y, double z)

- points: This the pointer to the collection of points created by a previous call to sdCreateSdPoints().
- x: The value on the X axis.
- y: The value on the Y axis.
- z: The value on the z axis. If the graph is two dimensional then this value is set to zero.
5.2.3 sColPoint()

The actual coloring of the points is done by the call:

```c
int sColPoint(int sock, double time, unsigned char tid, SdPoints *points, char *color)
```

- **sock**: This is the handle to the server returned by a prior `sGetServer()` call.
- **time**: The simulation time at which this call is made.
- **tid**: The task identifier returned by a prior `sdCreateTask()` call.
- **points**: The SdPoints structure created and filled as explained above.
- **color**: The color in which these points must be draw specified as a color name.

5.2.4 sdFreeSdPoints()

After a collection is used and no longer needed it can be freed using the call:

```c
void sdFreeSdPoints(SdPoints *points)
```

- **points**: This the pointer to the collection of points created by a previous call to `sdCreateSdPoints()`.

5.2.5 sPltPoint() and sPltPoint2D()

Lines can be plotted between pairs of points provided to the server in sequence. Plotting starts when the second point is specified and continues with the specification of each additional point by a call to the function:

```c
int sPltPoint(int sock, double time, unsigned char tid, unsigned int pid,
               double x, double y, double z, char *color)
```

- **sock**: This is the handle to the server returned by a prior `sGetServer()` call.
• **time**: The simulation time at which this call to the Server is made.

• **tid**: The task identifier returned by a prior `sdCreateTask()` call.

• **pid**: The value of the point identifier. Each task may have several lines plotted and this value identifies the line that it refers to. The programmer decides the values of these identifiers and they may be simply numbered from 0, 1, 2, ... .

• **x**: The value on the X axis.

• **y**: The value on the Y axis.

• **z**: The value on the z axis. If the graph is two dimensional then this value is set to zero.

• **color**: The color in which the line is drawn specified as a color name.

To plot a 2D line a convenience function is available:

```c
int sPltPoint2D(int sock, double time, unsigned char tid, unsigned int pid,
                double x, double y, char *color)
```

### 5.2.6 `sRestartPlot()`

To restart the plot from a value different from the one that is specified in a previous call to `sPltPoint()` the function `sRestartPlot()` may be used.

```c
int sRestartPlot(int sock, double time, unsigned char tid, unsigned int pid,
                 double x, double y, double z)
```

• **sock**: This is the handle to the server returned by a prior `sGetServer()` call.

• **time**: The simulation time at which this call to the Server is made.

• **tid**: The task identifier returned by a prior `sdCreateTask()` call.

• **pid**: The value of the point identifier. Each task may have several lines plotted and this value identifies the line that it refers to. The programmer decides the values of these identifiers and they may be simply numbered from 0, 1, 2, ... .
• **x**: The value on the X axis.
• **y**: The value on the Y axis.
• **z**: The value on the z axis. If the graph is two dimensional then this value is set to zero.

### 5.3 Histograms

Histogram tasks may be created by specifying the `HISTTASK` type in the `sdCreateTask()` call. Support is provided for computing automatically the frequency given the x value. The frequency computed may be used to plot the histogram if required. An `SdXaxisInterval` parameter must be specified for a histogram task, besides the axis related parameters.

#### 5.3.1 `sdFreqHistPoint()`

To compute the histogram frequency use may be made of the function:

```c
double sdFreqHistPoint(unsigned char tid, double x, double *f)
```

• **tid**: The task identifier returned by a previous call to `sdCreateTask()`.
• **x**: The x value for which the cumulative frequency is to be computed.
• **f**: The address of a location where the corresponding computed frequency may be stored.

#### 5.3.2 `sHistPoint()`

To actually draw a line bar corresponding to a given x value and its frequency use the function `sHistPoint()`.

```c
int sHistPoint(int sock, double time, unsigned char tid,
               double x, double y, char *color)
```

• **sock**: This is the handle to the server returned by a prior `sGetServer()` call.
• **time**: The simulation time at which this call to the Server is made.
• **tid**: The task identifier returned by a prior sdCreateTask() call.
• **x**: The value on the X axis.
• **y**: The frequency.
• **color**: The line bar is displayed in this color.

### 5.4 Displaying the Level of a Variable

To display the level of a variable a simple Scale is used. It also displays the name of the variable along with its current value. The corresponding task type is LEVELTASK. The orientation of the level may be horizontal or vertical.

#### 5.4.1 sDisplayLevel()

To display the level of variable after the corresponding task is created the sDisplayLevel() function may be called.

```c
int sDisplayLevel(int sock, double time, unsigned char tid, double value)
```

• **sock**: This is the handle to the server returned by a prior sGetServer() call.
• **time**: The simulation time at which this call to the Server is made.
• **tid**: The task identifier returned by a prior sdCreateTask() call.
• **value**: The current value of the variable.

### 5.5 Building a network

To build a network the DISplay library provides the GRAPHTASK type. Nodes made be added dynamically and these may be connected with arcs. The colors of the arcs and nodes may also be changed. It is also possible to link the arc or the node with application specific dialogs that are run when the node or the arc is clicked. Functions are provided to manage a Graph and include those for adding, deleting and changing parameters of nodes and arcs.
5.5.1 sGAddNode()

To add a node to a Graph that has been created:

    int sGAddNode(int sock, double time, unsigned char tid, int *node_num
                   char *label, double min, double max,
                   unsigned did_from, unsigned did_to, char *panel_label)

- **sock**: This is the handle to the server returned by a prior `sGetServer()` call.
- **time**: The simulation time at which this call to the Server is made.
- **tid**: The task identifier returned by a prior `sdCreateTask()` call.
- **node_num**: A unique node number that is generated by the call. This must be used when subsequent changes may be made to the node. The call fills in a value, so an address must be passed.
- **label**: The label on the node.
- **min**: The minimum value that is associated with this node. As the value changes the node is shown in a different color ranging from different shades of black to white for the max value.
- **max**: The maximum value associated with the node.
- **did_from**: The identifier of the first dialog in a group that is associated with this node.
- **did_to**: The identifier of the last dialog in a group that is associated with this node.
- **panel_label**: The label of the panel is popped up when this node is clicked.

5.5.2 sGAddArc()

To add an arc to a Graph that has been created:

    int sGAddArc(int sock, double time, unsigned char tid, int *arc_num
                 char *label, double min, double max,
                 unsigned node_from, unsigned node_to,
                 short direction, short line_style, short line_width,
                 unsigned did_from, unsigned did_to, char *panel_label)
- sock: This is the handle to the server returned by a prior sGetServer() call.
- time: The simulation time at which this call to the Server is made.
- tid: The task identifier returned by a prior sdCreateTask() call.
- arc_num: A unique arc number that is generated by the call. This must be used when subsequent changes may be made to the arc. Pass the address here and the call will fill in the value.
- label: The label on the node.
- min: The minimum value that is associated with this arc. As the value changes the arc is shown in a different color ranging from different shades of black to white for the max value.
- max: The maximum value associated with the arc.
- node_from: This the node number of the node from which to draw the arc.
- node_to: This the node number of the node to which to draw the arc.
- direction: This is a DISplay define constant and is described in table 3.
- line_style: Defined in table 3.
- line_width: Specify the line width as an integer.
- did_from: The identifier of the first dialog in a group that is associated with this node.
- did_to: The identifier of the last dialog in a group that is associated with this node.
- panel_label: The label of the panel is popped up when this node is clicked.

5.5.3 sGChangeVal()

To change the values of a Graph node or arc use:

```c
int sGChangeVal(int sock, double time, unsigned char tid, int num
    int node_or_arc, short line_style, short line_width, double val);
```
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDBIDIRECTED</td>
<td>A Bidirected arc</td>
</tr>
<tr>
<td>SDDIRECTED</td>
<td>A Uni-directed arc</td>
</tr>
<tr>
<td>SDUNDIRECTED</td>
<td>A Non-directed arc</td>
</tr>
<tr>
<td>SDLINESOLID</td>
<td>Solid Line</td>
</tr>
<tr>
<td>SDLINEONOFFDASH</td>
<td>A Dashed Line</td>
</tr>
<tr>
<td>SDLINEDOUBLEDASH</td>
<td>A double dashed line</td>
</tr>
<tr>
<td>SDVERTICAL</td>
<td>Vertical Orientation</td>
</tr>
<tr>
<td>SDHORIZONTAL</td>
<td>Horizontal Orientation</td>
</tr>
<tr>
<td>SDGNODE</td>
<td>Identifies a node</td>
</tr>
<tr>
<td>SDGARC</td>
<td>Identifies an arc</td>
</tr>
</tbody>
</table>

Table 3: Values for Graph Related Calls

- **sock**: This is the handle to the server returned by a prior `sGetServer()` call.
- **time**: The simulation time at which this call to the Server is made.
- **tid**: The task identifier returned by a prior `sdCreateTask()` call.
- **num**: A unique arc or node number that is generated by the Add call.
- **node_or_arc**: This is a DISplay constant and define in table 3.
- **line_style**: Defined in table 3.
- **line_width**: Specify the line width as an integer.
- **val**: The value that is used in changing the color of the node or arc.

### 5.5.4 `sGDelNode()` and `sGDelArc()`

To delete a node or an arc use the following functions:

```c
int sGDelNode(int sock, double time, unsigned char tid, int node_num)

int sGDelArc(int sock, double time, unsigned char tid, int arc_num)
```
6 Miscellaneous functions

There are two functions that may be found useful in certain application.

6.1 sNullMsg()

To advance the time at the Server in the absence of genuine message the following function may be used. It has no effect other than advancing the time that the Server associates with each process that is connected. In a single process situation this function need never be called.

```c
int sNullMsg(int sock, double time)
```

- `sock`: This is the handle to the server returned by a prior `sGetServer()` call.
- `time`: The simulation time at which this call to the Server is made.

6.2 sStrPrintMsg()

The other function is to print textual messages in the display window. This may be useful for a variety of purposes including writing out results, displaying information, etc.

```c
int sStrPrintMsg(int sock, double time, char *format, ...)
```

- `sock`: This is the handle to the server returned by a prior `sGetServer()` call.
- `time`: The simulation time at which this call to the Server is made.
- `format`: This the format that is specified in a standard `printf()` call. It may be followed by the variables that are to be printed as per the specified `format`. 
Dialogs are the basic mechanism through which user interaction is implemented. A dialog is defined as a "form" comprising of a set of values, and associated with each value is a set of items which describe that value. In the current implementation, this set of items is limited to a textual description of the item and a specification describing the type of the value. The textual description can also be used as a question to the user to input a value of the required type. Several related dialogs may be made part of a single "panel" from which the user can point and click the label of the particular dialog that is being invoked. On being invoked the current state of the dialog is displayed. Dialogs allow for two way interaction between the application and the user and are of four types, depending on the direction in which interaction is allowed or restricted, and who invokes the dialog. Table 4 shows further how the dialogs may be classified.

1. User Query Dialog (SDUSER_QUERY) This is a dialog in which the request for information is sent from the DS to the application. The application handles the query and returns the required values. Invocation is done by the user, who may fill in some of the values in the dialog and the application may use these for its computation and return the remaining computed values.

2. User Command Dialog (SDUSER_COMMAND) These allow the user to send information to the application asynchronously. The application must be willing to handle these dialog messages through appropriate user written handlers which are registered with the dialog when it was created.

3. Model Query Dialog (SDMODEL_QUERY) This is similar in functionality to a remote call. The application pops up the dialog at the DS at predefined points and then waits for the user to reply. The application is blocked until the user reply comes in.
4. **Model Display Dialog (SDMODEL_DISPLAY)** This is a dialog where the information is passed to the DS at predefined points in the application. The server collects the information and displays it only when the user invokes the dialog by clicking on the associated button. The information can be received only at those points at which the application sends it. A special form of this dialog type is one that is associated with a task (SDDISP_TASK). The associated task is popped up when the panel button is clicked.

It is the model developers' responsibility to provide for functions called "dialog handlers" at the application which will handle replies or queries from the user. These functions can be registered when the dialog is created and are automatically called when a dialog message of the given type is received. If asynchronous dialogs are used then the simulation developer must also arrange to call a function provided in the library from time to time, which will read dialog messages from the DS, and then call the application dialog handlers. An example given later in Section 8 shows how this may be done. Since all messages are logged it is possible to find out later, after the simulation is completed, as to the points of time when the interactions between the user and the application took place. This could be important while doing post run analysis of simulations.

7.1 **Dialog Management**

7.1.1 **The SdQtnAns Structure**

To define a dialog the dialog designer must write out the parts of the dialog using this structure. An example use of this structure is as follows:

```c
static SdQtnAns dialog_parm[] = {
    /* parameters of the simulation */
    {"Mean Inter-Arrival Time", SDFLOAT},
    {"Mean Service Time", SDFLOAT},
    {"Number of Servers", SDINT},
    {"Number of Customers", SDINT},
};
```

Each line in this structure defines the description of a "value" in the dialog. The line consists of a description of what the value means or an instruction to the user of what must be entered followed by a type for the value. For example, the first line above specifies that the value is the "mean inter-arrival time" and its type is SDFLOAT. All DISplay types are defined in 5.
<table>
<thead>
<tr>
<th>Type</th>
<th>C Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDNONE</td>
<td>void</td>
<td>The item in the dialog has no value. The associated string is displayed.</td>
</tr>
<tr>
<td>SDINT</td>
<td>int</td>
<td>Integer type is expected in the value.</td>
</tr>
<tr>
<td>SDFLOAT</td>
<td>float</td>
<td>Float type is expected in the value for this item.</td>
</tr>
<tr>
<td>SDSTRING</td>
<td>char *</td>
<td>A string is expected as the value.</td>
</tr>
<tr>
<td>SDTOGGLE</td>
<td>int</td>
<td>This is similar to a boolean, either 1 or 0.</td>
</tr>
<tr>
<td>SDHEADING</td>
<td>void</td>
<td>This can be used as an heading for a dialog. The associated variable has no value stored.</td>
</tr>
</tbody>
</table>

Table 5: Values for Types in Dialogs

7.1.2 sdCreateDialog()

The function returns a unique dialog identifier (did) which must be used in all future references to the dialog.

```
unsigned sdCreateDialog(char *label, unsigned type, unsigned tid, char *bcolor,
                        int num_items, SdQtnAns *sdQtnAns,
                        int (*handler_func)(char *reply[]))
```

- **label**: This is the dialog label that is displayed on the panel button.
- **type**: The type of the Dialog. Choose one depending on the types described above in Section 7.
  - **tid**: The task identifier returned by a prior sdCreateTask() call.
- **bcolor**: The background color of the displayed button.
- **num_items**: Is the number of items or values in the Dialog.
- **sdQtnAns**: Items are defined using the special structure that is described above.
- **handler_func**: Is the function at the client that handles any reply or command from the user. Replies are passed to the handler in the form of an array of replies.
7.1.3 sdCreateDialogTask() and sdCreateDialogNotask()

The first of these functions is used when defining a dialog related to a task and the second one when defining a true dialog that consists of a SdQtnAns structure. These are convenience functions and their interface is as follows:

unsigned sdCreateDialogTask(char *label, unsigned tid)

unsigned sdCreateDialogNotask(char *label, unsigned type, int num_items, SdQtnAns *sdQtnAns, int (*handler_func)(char *reply[]))

7.1.4 sBeginDialog()

This registers the dialog with the server. A group of dialogs can be registered at the same time. All these dialogs will be grouped and will be invoked from a single panel on the Server.

int sBeginDialog(int sock, double time, unsigned dialog_from, unsigned dialog_to, char *title)

- sock: This is the handle to the server returned by a prior sGetServer() call.
- time: The simulation time at which this call to the Server is made.
- dialog_from: This the did of the dialog starting from which grouping will start up to dialog_to.
- dialog_to: The did up and including which are grouped in the same panel displayed on the server.
- title: This will be the title of the panel and also the label of the interaction push-button that will be visible on the main interface window.
7.1.5 sChangeDialog()

A registered dialog can be changed to some extent. Its label, background color, the type and description of the values represented by the dialog can be changed dynamically. However the number of values that the dialog was setup with and the dialog type cannot be changed. The handler that handles the reply and the tid if it was a SDDISP_TASK type dialog can also be changed.

\[
\text{unsigned sChangeDialog(int sock, double time, unsigned did,}
\]

\[
\text{char *label, unsigned tid, char *bcolor,}
\]

\[
\text{int num_items, SdQtnAns *sdQtnAns,}
\]

\[
\text{int (*handler_func)(char *reply[]))}
\]

- sock: This is the handle to the server returned by a prior sGetServer() call.
- time: The simulation time at which this call to the Server is made.
- did: The dialog identifier for which the change is to be made.
- label: This is the dialog label that is displayed on the panel button.
- tid: The task identifier returned by a prior sdCreateTask() call.
- bcolor: The background color of the displayed button.
- num_items: Is the number of items or values in the Dialog. This must not change from the setup value.
- sdQtnAns: Items are defined using the special structure that is described above.
- handler_func: Is the function at the client that handles any reply or command from the user. Replies are passed to the handler in the form of an array of replies.

7.1.6 sdDeleteDialog()

This deletes the dialog at the client and frees up all areas that were used. This is to be done only when the program is to finish.

\[
\text{int sdDeleteDialog(unsigned dialog_from, unsigned dialog_to)}
\]

- dialog_from: This the did of the dialog starting from which deletion will start upto dialog_to.
- dialog_to: The did up and including which are deleted.
7.2 Handling Queries and Replies

DISplay provides support to the user for creating reply structures to user queries. Creation of these structures is based on the did that is returned from the sdCreateDialog() call. Each reply is basically an array of strings that represent the values that are filled in either by the user or the application. Answers that are received by the application from the user are received as an array of strings which must be converted to the correct types using standard functions provided by the standard libraries.

7.2.1 sdCreateReply() and sdFreeReply()

This function creates a reply structure to a user query. The reply structure is basically an array of strings which stores the values of any DISplay type. When the reply structure is no longer needed it may be freed using the function provided.

```
char **sdCreateReply(unsigned did)
```

- did: The dialog identifier for which a initialized reply is made.

```
char **sdFreeReply(unsigned did)
```

- did: The dialog identifier for which the initialized reply is freed.

7.2.2 sdAddToReply()

The user can add to the created reply structure one at a time. The index of the reply in the dialog must be passed as a parameter.

```
char **sdAddToReply(unsigned did, int i, <type> value)
```

- did: The dialog identifier for which a initialized reply is made.
- i: This is the index of this reply or value in the dialog form. Indexes start from zero.
- value: Actual values or variable may be used. The value must be of the <type> specified when the dialog was created.
7.2.3 sQueryReply()

This function is used for a MODEL_QUERY type dialog to query the user and obtain a reply. The application waits until the query is answered. The reply is available when the function returns. The time variable is filled in with the simulation time at which the user replied.

```
int sQueryReply(int sock, double *time, unsigned did, char *reply[])
```

- **sock**: This is the handle to the server returned by a prior sGetServer() call.
- **time**: The simulation time at which the query was sent. The return value contains the time that the user replied.
- **did**: The dialog identifier for which the query is made.
- **reply**: The replies are available as an array of strings. The application may look at each array element and process it depending on its type.

7.2.4 sHandleReply()

This function is the primary means by which user queries or commands may be picked up by the application asynchronously and the appropriate handler executed. The programmer must arrange for this call to be made at a reasonable frequency. A time for which the call must block before returning can also be specified. If this time is specified as zero then the call returns immediately if no messages are available. If several messages are available they are all read in and the handlers called before control is returned.

```
int sHandleReply(int sock, lon sec, long msec)
```

- **sock**: This is the handle to the server returned by a prior sGetServer() call.
- **sec**: The number of seconds to wait for a message before returning.
- **msec**: The number of microseconds to wait in addition to the sec before returning.
7.2.5  sReplyDispMsg()

This function is used for a USER_QUERY type dialog to reply to the user with a reply created by the sdCreateReply() call.

```c
int sReplyDispMsg(int sock, double time, unsigned did, char *reply[])
```

- **sock**: This is the handle to the server returned by a prior sGetServer() call.
- **time**: The simulation time at which the reply was sent.
- **did**: The dialog identifier for which the reply is made.
- **reply**: The replies are available as an array of strings. At the server the replies are displayed in the dialog form if the window is visible to the user.

8  Example

This is an example of a CSIM program simulating the M/M/n server. The example creates and uses several tasks and dialogs. There is one include file that must be included in every program that uses the DISplay library. This is the sdclnt.h file. The file sdclnt.h also internally includes the files sdconsts.h and sdclconsts.h.

```c
/* simulate an M/M/n queue */
#define DISPLA
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <csim.h>

/* Displa Include Files */
#include <sdclnt.h>

#define SVTM 4.0  /* mean of service time distribution */
float svtm;
#define IATM 2.0  /* mean of inter-arrival time distribution */
```
float iatm;
#define NARS 10000 /*number of arrivals to be simulated*/
int nars; /* 50000 takes 100000 */

FACILITY f; /*pointer for facility*/
EVENT done; /*pointer for counter*/
TABLE tbl; /*pointer for table*/
QTABLE qtbl; /*pointer for qhistogram*/
int cnt; /*number of active tasks*/

#define NS 5 /*number of servers*/
int ns;

#ifdef DISPLA
void check_changes(/*void*/);
static int handle_server_change(/*char *reply[]*/);
static int handle_arrival_change(/*char *reply[]*/);
int connect_ds(/*char *argv[]*/);
char *host, *service;

define Dialogs */
static SdQtnAns dialog_parm[] = { /* parameters of the simulation */
    { "Mean Inter-Arrival Time", SDFLOAT },
    { "Mean Service Time", SDFLOAT },
    { "Number of Servers", SDINT },
    { "Number of Customers", SDINT },
};

static SdQtnAns dialog_ns[] = { /* the Server change dialog */
    { "Change Number of Servers to:", SDINT },
    { "Old Number of Servers was:", SDINT },
};

static SdQtnAns dialog_arr[] = { /* the arrival rate change dialog */
    { "Change Inter-Arrival to:", SDFLOAT },
    { "Old Arrival Rate was:", SDFLOAT },
};

static int ns_dialog_id;
static int arr_dialog_id;
static int parm_dialog_id;
static int queue_task, util_task, qtabl_task;
static int util_dialog, queue_dialog, qtabl_dialog;
#endif

sim(argc, argv)
    /*1st process - named sim */
    int argc;
    char *argv[];
{
    int i;
#endif

32
if (argc != 3) {
    printf("Usage:%s host port\n",argv[0]);
    return(1);
}
sock = ds(argv);
if (sock == SDERROR)
    return(1);
get_parameters();
#ifif
if (iatm == 0.0)
    iatm = IATM;
if (svtm == 0.0)
    svtm = SVTM;
if (ns == 0)
    ns = NS;
if (nars == 0)
    nars = NARS;
set_model_name("M/M/n Queue");
create("sim");
f = facility_ms("facility",ns);
done = event("done");
tbl = table("resp tms");
qttbl = qhistogram("num in sys", 10);
cnt = nars;
iatm = IATM;
for(i = 1; i <= nars; i++) {
    hold(expntl(iatm));
    cust();
}
wait(done);
report();
mdlstat();
#else DISPLA
    check_changes();
#endif DISPLA
}
wait(done);
report();
mdlstat();
#endif DISPLA
sEndTask(sock, clock, util_task);
sEndTask(sock, clock, queue_task);
sStrPrintMsg(sock, clock, "Simulation Completed at time %lf", clock);
#endif DISPLA
cust() {
    float t1;
    create("cust");
t1 = clock;
note_entry(qtbl);
#endif DISPLA
plot_results();
```c
#ifdef DISPLA
int ds(argv)
  char *argv[];
{
  char *simname = "M/M/n Queue";

  sock = sGetServer(simname, argv[1], argv[2]);
  if (sock == SDERROR) {
    fprintf(stderr, "Could not connect to display server;%s:%s\n", argv[1], argv[2]);
    return(sock);
  }

  sNewProcessMsg(sock, TRUE, FALSE, 0);
  if (sock == SDERROR) {
    printf("Could not connect to display server\n");
    return (sock);
  }

  util_task = sdCreateTask(SDPLPTSMSG, TRUE, FALSE, FALSE);
  sdSetTaskValues(util_task,
                   SdXaxisName, "Time", SdYaxisName, "Utilization",
                   SdTitle, "Facility Utilization",
                   SdXmin, 0.0, SdXmax, 100000.0,
                   SdYmin, 0.0, SdYmax, 10.0,
                   SdNoPoints, 1,
                   SdDim, 2,
                   NULL);
  sBeginTask(sock, clock, util_task);

  queue_task = sdCreateTask(SDPLPTSMSG, TRUE, FALSE, FALSE);
  sdSetTaskValues(queue_task,
                  SdXaxisName, "Time", SdYaxisName, "Queue Size",
                  SdTitle, "Facility Queue Size",
                  SdXmin, 0.0, SdXmax, 100000.0,
                  SdYmin, 0.0, SdYmax, 150.0,
                  SdNoPoints, 2,
                  SdDim, 2,
                  NULL);
  sBeginTask(sock, clock, queue_task);

  qtabl_task = sdCreateTask(SDHISTPTMSG, TRUE, TRUE, FALSE);
  sdSetTaskValues(qtabl_task,
```
void check_changes()
{
    /* read any messages from the display server */
    (sHandleReply(sock, 0, 0));
}

static int handle_server_change(reply)
    char *reply[];
{
    static int i = 0;
    char buf[16];
    int new_ns;
    char **send_reply;
    if ((new_ns = atoi(reply[0])) == 0) || (new_ns == ns))
        return(0);
    wait(event_list_empty);
    sprintf(buf, "facility \%d", ++i);
    f = facility_ms(buf, new_ns);
    sStrPrintMsg(sock, clock, "New facility created at time \%f\n", clock);
    send_reply = sdCreateReply(ns_dialog_id);
    sdAddToReply(ns_dialog_id, 0, new_ns);
    sdAddToReply(ns_dialog_id, 1, ns);
    ns = new_ns;
static int handle_arrival_change(reply)
char *reply[];
{
  float new_iatm;
  char **send_reply;
  if ((new_iatm = (float)atof(reply[0])) == 0) || (new_iatm == iatm))
    return(0);
  wait(event_list_empty);
  sStrPrintMsg(sock, clock, "Changing inter-arrival time at time %f\n", clock);
  send_reply = sdCreateReply(arr_dialog_id);
  sdAddToReply(arr_dialog_id, 0, new_iatm);
  iatm = new_iatm;
  sReplyDispMsg(sock, clock, arr_dialog_id, send_reply);
  return(1);
}

int plot_results()
{
  int ql;
  double xs, ys;
  ql = qlength(f);
  sPltPoint(sock, clock, util_task, 0, clock, (double)util(f), 0.0, "black");
  sPltPoint(sock, clock, queue_task, 0, clock, (double)ql, 0.0, "black");
  sPltPoint(sock, clock, queue_task, 1, clock, (double)qlen(f), 0.0, "blue");
  ys = sdFreqHistPoint(qtabl_task, (double) (ql+num_busy(f)), &xs);
  sHistPoint (sock, clock, qtabl_task, xs, ys, "black");

  get_parameters()
  {
    char *reply[4];
    double dbl, timereply;
    int ivl;
    timereply = clock;
    sQueryReply(sock, &timereply, parm_dialog_id, reply);
    if ((dbl = atof(reply[0])) > 0.0)
      iatm = dbl;  /* Inter-arrival Time */
    if ((dbl = atof(reply[1])) > 0.0)
      svtm = dbl;  /* service time */
    if ((ivl = atoi(reply[2])) > 0)
      ns = ivl;   /* number of servers */
    if ((ivl = atoi(reply[3])) > 0)
nars = ivl; /* number of customers */
sStrPrintMsg(sock, clock, "M/M/n parameters");
sStrPrintMsg(sock, clock, "Service Time %f Inter-Arrival %f Servers %d", svtm, iatm, ns);
}

#endif