

The GTK+ Drag-and-Drop Mechanism

1 Overview

Drag-and-drop (*DND*, for short) is an operation in applications with graphical user interfaces by which users can request, in a visual way, that running applications exchange data with each other. To the user, in a drag-and-drop operation, it appears that data is being dragged from the *source* of the drag to the *destination* of the drag. Because applications are independent of each other and written without cognizance of who their partners will be in a DND operation, for DND to work, there must be underlying support by either the operating system or the windowing system.

The Mac OS operating system has always had built-in support for drag-and-drop. Microsoft Windows did not have it; it was added on top of the operating system in Windows 95 and later. UNIX has no support for it at all because the graphical user interfaces found in UNIX systems are not part of the operating system. Support for DND is provided by the X Window system.

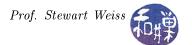
Over the years, several different protocols were developed to support DND on X Windows. The two most common were Xdnd and Motif DND. GTK+ can perform drag-and-drop on top of both the Xdnd and Motif protocols.

In GTK+, for an application to be capable of DND, it must first define and set up the widgets that will participate in it. A widget can be a source and/or a destination for a drag-and-drop operation. A source widget is one that can provide drag data, so that the user can drag something off of it. A destination widget is one that can receive drag data. Destination widgets can limit from whom they will accept drag data., e.g. the same application or any application (including itself). Destination widgets can also define the types of data that they are willing to receive.

In GTK+, DND is made possible through the use of the signals emitted by widgets, i.e., the signals defined for the GtkWidget base class. There are several signals that are emitted during the various stages of a DND operation. The most fundamental ones are those involved in the transfer of data from the source to the destination. This is, after all, the whole point of DND – to make this transfer happen. Understanding how this transfer happens and the role that signals play in carrying it out is crucial to being able to write programs that use drag-and-drop. Therefore, we begin by describing the steps involved in the transfer.

The actual transfer begins when the user, having started a drag and holding the mouse button down, releases it over a potential destination widget.

- 1. At this moment a "drag-drop" signal is emitted on this widget.
- 2. The "drag-drop" signal causes two simultaneous events:
 - (a) If no errors occurred, a "drag-data-get" signal is emitted on the source widget. This signal is in essence a request for data from the source.
 - (b) If a handler for the drag-drop signal was connected to the destination widget, then that handler runs.
- 3. In response to "drag-drop" signal, the destination widget has to indicate that it wishes to receive data from the source of the drag; it does this by calling gtk_drag_get_data().
- 4. If a handler for the "drag-data-get" signal was connected to the source widget, when the source receives this signal, it must deliver the data. It does this by calling gtk_selection_data_set(). This function copies the data into a GtkSelection object.



- 5. When the source widget has copied its data into a GtkSelection object, GTK arranges for a "dragdata-received" signal to be emitted on the destination widget¹.
- 6. If a handler for the "drag-data-received" signal was connected to the destination widget, that handler's argument includes the selection object containing the data, and the destination can copy the data out of the object into its own variables.

This may seem complicated on first reading, but the basic idea is that the communication between the source and destination takes place through a third entity, GTK+'s *selection* mechanism, which will be explained in detail below, and is made possible by the underlying DND protocol. Notice that neither the source nor the destination know the others identity.

GTK+ provides a way to do very basic DND without diving into the complexity of the topic, if you are willing to accept default behaviors and do not need to drop many different types of data. On the other hand, if you want to do things such as customizing the drag icon on the start of a drag, deciding whether or not to accept drag data depending no cursor position, or deciding what type of data to accept based on the cursor's position on the destination, highlighting the widget when it is a potential drop site, checking error conditions, and so on, then you need to learn how to use a larger portion of the API.

These notes describe how to do many of these tasks. The remaining sections provide the background and detailed information required to implement drag-and-drop in elementary and more advanced ways. We begin by covering background material.

2 Background

2.1 Atoms

A GdkAtom is a fundamental type in GDK, its significance arising from the fact that it is an efficient way to represent large chunks of data. Windows under X can have any number of associated properties attached to them. Properties in general are arbitrary chunks of data identified by atoms. In X, an atom is a numeric index into a string table on the X server. They are used to transfer strings efficiently between clients without having to transfer the entire string. A property has an associated type, which is also identified using an atom.

Every property has an associated format, which is an integer describing how many bits are in each unit of data inside the property. It must be 8, 16, or 32. For example, if a property is a chunk of character data, then its format value would be 8, the number of bits in a character. If it is an integer, its format would be 32.

GDK provides functions for manipulating atoms. These will be needed when implementing drag-and-drop in GTK+. Your application will need to *intern* various strings. To *intern* a string means to store it in an internal table used by GDK and obtain an atom that identifies it for later access. The function to intern a string is

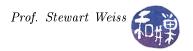
GdkAtom gdk_atom_intern

(const gchar *atom_name, gboolean only_if_exists);

This makes a copy of the string to be interned, i.e., the name of the atom, and returns an atom for that string. The second argument is ignored by GDK. If the string already exists, it returns its atom. If it does not, it creates a new atom.

The inverse function is

¹Actually, when the widget called gtk_drag_get_data(), that function caused emission of a signal on the selection object. The selection object's signal handler for that signal was synchronized by the underlying DND protocol (e.g. X11) and when the data was actually made available in the selection object, the selection object emitted the "drag-data-received" signal on the destination widget.



gchar * gdk_atom_name

(GdkAtom atom);

Given an atom, this returns a newly-allocated string containing the string corresponding to atom. Your application must release the memory with $g_free($) when it is finished with it.

You will not need to use any of the other functions related to atoms for DND.

2.2 Selections

The *selection mechanism* provides the basis for different types of communication between processes. In particular, drag-and-drop and the GtkClipboard work because of the selection mechanism. The GtkSelectionData object is used to store a chunk of data along with the data type and other associated information. In drap-and-drop, the term *selection* refers to the choice of data that is supplied and/or received by a widget. When a drop is made on a widget, there may be several different types of data in the selection object that is provided to it; the widget has to decide which type of data it wants to accept. Therefore, one says that the widget *selects* a particular chunk of data.

The GtkSelectionData object acts like the medium of transport between drag sources and drag destinations. The source will use one of various methods in the GtkSelectionData class to describe the types of data that it offers, and the destination widgets will use methods to search the selection for data types that interest them. When data is actually transferred, the selection object will be used as the intermediary between the two widgets.

2.3 Targets

The word "target" is a bit misleading in the context of DND. Although it sounds like it means the "target of a drop", it does not. The word "destination" refers to this widget. To avoid any confusion, we will *never* use the word "target" to mean the destination. A target is a type of data to be used in a DND operation. For example, a widget can supply a string target, an image target, or a numeric target. Targets are represented by the GtkTargetEntry structure.

The GtkTargetEntry structure represents a single type of data than can be supplied by a widget for a selection or received by a destination widget in a drag-and-drop operation. It consists of three members: (1) target, a string representing the type of data in a drag, (2) flags, a set of bits defining limits on where the target can be dropped, and info, an application assigned integer ID.

```
typedef struct {
   gchar *target;
   guint flags;
   guint info;
} GtkTargetEntry;
```

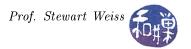
The target string is provides a human-understandable description of the data type. It is important to use common sense target names, because if your application will accept drags or offer data to other applications, the names you choose should be those other applications might use also. The info member serves to identify the target in the functions that access and manipulate target data, because integers allow for faster look-ups and comparisons.

The flags value may be one of the following:

GTK_TARGET_SAME_APP The target will only be selected for drags within a single application.

GTK_TARGET_SAME_WIDGET The target will only be selected for drags within a single widget.

GTK_TARGET_OTHER_APP The target will not be selected for drags within a single application.



GTK_TARGET_OTHER_WIDGET The target will not be selected for drags withing a single widget.

If flags == 0, it means there are no constraints.

Usually you would create an enumeration within the application to provide meaningful names for the info values, for example:

```
typedef enum
{
    TEXT_HTML,
    STRING,
    IMAGE_JPEG,
    NUMBER,
    TEXT_URI,
    N_TARGETS
} target_info;
```

Using this enumeration we could define a few different targets as follows:

```
GtkTargetEntry string_target = {"string_data", 0, STRING};
GtkTargetEntry html_target = {"text/html", GTK_TARGET_SAME_APP, TEXT_HTML};
GtkTargetEntry image_target = {"image/jpeg", GTK_TARGET_SAME_WIDGET, IMAGE_JPEG};
```

The string_target and the html_target both represent text, but the latter would identify itself to a destination widget was capable of parsing the HTML and preferred receiving it over plain text. Such a widget would probably select the html_target rather than the string_target. The image_target could be used for JPEG image formats. The string target has no flags and therefore no limits on where it can be dropped. The html_target is only allowed to be dropped into the same application as the source widget, and the image_target is constrained to be dropped into the same widget.

2.4 Target Tables and Target Lists

A target table is an array of type GtkTargetEntry. There is no object specifically declared to be a target table. It is just understood that it is an array of target entries. Target tables are useful in application code for consolidating target entry definitions. More importantly, the function that sets up a widget as a DND source widget, gtk_drag_source_set(), requires the set of targets to be passed to it as a table. Target tables can also be passed as arguments to certain other functions related to GtkSelectionData. The following is an example of a target table:

```
GtkTargetEntry target_entries[] = {
    {"text/html", 0, TEXT_HTML },
    {"STRING", 0, STRING},
    {"number", 0, NUMBER},
    {"image/jpeg", 0, IMAGE_JPEG},
    {"text/uri-list", 0, TEXT_URI}
};
```

A target list is not a list of GtkTargetEntry structures, as you might expect. It is a list of GtkTargetPair structures, and it serves a different purpose from target tables. A GtkTargetPair is a internal data structure used by GTK+. It is defined by



```
struct GtkTargetPair {
  GdkAtom target;
  guint flags;
  guint info;
};
```

Notice that it differs from a GtkTargetEntry in a single respect: it uses a GdkAtom instead of a character string to identify the target. Recall from Section 2.1 above that a GdkAtom is an integer that GDK uses to represent a string internally; it is the index into an array of strings. An atom is only defined when a string is "interned."

The functions that take a GtkTargetEntry and store that target for later use intern the character string and create an atom for it. Once this has been done, that target can be represented by a GtkTargetPair. In other words, the target atom in the GtkTargetPair represents a target that has already been defined in some GtkTargetEntry.

Because atoms make for faster comparison and identification and save storage space, target lists are more efficient than target tables and are used more extensively than them by GTK+. There are methods in the GtkSelectionData class for going back and forth between target table and target list representations of the targets. For example:

gtk_target_list_new() creates a target list from a target table

gtk_target_list_add_table() prepends a target table to an existing target list

gtk_target_table_new_from_list() creates a target table that contains the same targets as the given list.

Many of the methods provided by the GtkSelectionData class expect and manipulate target lists. They are of fundamental importance in using drag-and-drop, and we will have more to say about them below.

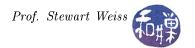
3 Signals Involved in Drag-and-Drop

Various signals come into play during a DND operation. Some are essential to handle and others are not. All signals are emitted on GTK+ widgets and their descriptions can be found in the API documentation of the GtkWidget class. The following table lists all of these signals, indicating whether it is emitted on the course or the destination, and what its purpose is.

Signal	\mathbf{Widget}	Purpose	
drag-begin-event	source	notifies source that drag started	
drag-motion	destination	notifies destination about drag pointer motion	
drag-drop	destination	notifies destination that data has been dropped	
drag-data-get	source	request for drag data from source	
drag-data-received	destination	source has sent target the requested data	
drag-data-delete	source	${ m source\ should/can\ delete\ data}$	
drag-end-event	source	notifies source that drag is done	
drag-failed	source	notifies source that drag failed	
drag-leave	destination	notifies destination that cursor has left widget	

3.1 The Typical Sequence of Events

The sequence of events that take place in a drag-and-drop is well-defined. The typical sequence is described below. Under certain conditions there will be slight deviations from it.



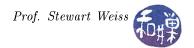
- 1. Everything begins when the user presses the mouse button over a source widget and starts a drag. At that moment, the "drag-begin-event" signal is emitted on the source.
- 2. When the mouse is on top of a destination widget, the "drag-motion" signal is emitted on that widget. This signal can be connected to the destination for various reasons. For one, you can use it to highlight the widget when the cursor is over it and the drag format and action are acceptable to the widget. For another, if only certain parts of the widget are drop zones, the handler is needed in order to determine whether the cursor is in a drop zone or not. If it is not in a drop zone, the handler should return FALSE and take no other action. Otherwise, it should display visual feedback to the user by calling gdk_drag_status() and return TRUE. Sometimes a drag-motion handler cannot decide whether the offered data is acceptable from the cursor position and data type, and must actually examine the data to know. In this case it will do the work typically done in a drag-drop handler. The details about this and other issues to be handled in a "drag-motion" handler are explained below.

If when you set up the destination widget using gtk_drag_dest_set(), you set any of the flags GTK_DEST_DEFAULT_DROP, GTK_DEST_DEFAULT_MOTION or GTK_DEST_DEFAULT_ALL on the widget, you will not be able to use the drag-motion signal this way, because GTK+ will handle it with its internal functions instead.

The "drag-motion" signal will be delivered to the widget each time that the cursor moves over the widget. If you want to detect when it enters and leaves the widget, you have to make use of the "drag-leave" signal, which is emitted on a destination widget whenever the cursor leaves it. An entry event takes place when it is the first "drag-motion" signal to be received after a "drag-leave" or the first one to be received. The handlers for the two signals can be coded to detect these conditions.

- 3. When the user releases the mouse button over the destination, the "drag-drop" signal is emitted on the destination widget. This signal should be connected to a signal handler whose primary objective is to determine whether the cursor position is in a drop zone or not, and if it is, to issue a request for the data from the source by calling gtk_drag_get_data() and return TRUE. If the cursor is not in a drop zone, it should return FALSE and take no other action.
- 4. When the destination issues a request for the source's data, whether in the drag-drop handler or the drag-motion handler, the "drag-data-get" signal will be emitted on the source. A handler for this signal should be connected to the signal. This handler is responsible for packing up the data and setting it into a selection object that will be available to the destination.
- 5. Once the source widget's drag-data-get handler has returned, the "drag-data-received" signal will be emitted on the destination. This signal should be connected to a signal handler on the destination widget. If the data was received in order to determine whether the drop will be accepted (as when the drag-motion handler requested the data), the handler has to call gdk_drag_status() and not finish the drag. In most cases, if the data was received in response to a "drag-drop" signal, the handler has to retrieve the data from the selection object and then call gtk_drag_finish(). If the drag was a move (the GdkDragAction was set to GDK_ACTION_MOVE in the source or destination), then in the call to gtk_drag_finish() it needs to pass a flag indicating that the data should be deleted in the source.
- 6. The call to gtk_drag_finish() causes a "drag-end" signal to be emitted on the source widget. It can connect a handler to this signal to do any post-processing needed after the drag. It will also cause a "drag-data-delete" signal to be emitted on the source if the destination passed the flag when it called gtk_drag_finish(). The source has to delete the data in its handler for this signal.

This normal sequence might not be followed if there was a failure at some point. In this case, the "drag-failed" signal will be emitted on the source. A handler can be attached to the source to deal with the failure, such as by logging a message.



4 Setting Up a Source Widget

A widget is set up as a source widget for drag operations by calling the function gtk_drag_source_set() on it. The prototype is

The first argument, widget, is the widget to be the drag source. The remaining arguments have the following meaning.

start_button_mask the bitmask of buttons that can start the drag, of type GdkModifierType.

targets the table of targets that the drag will support, which may be NULL.

n_targets the number of items in targets.

actions the bitmask of possible actions for a drag from this widget.

The values of the GdkModifierType enumeration are listed in the API documentation for the GdkWindow. The values have names such as GDK_BUTTON1_MASK, GDK_BUTTON2_MASK, and so on. In addition, you can bitwise-or modifiers such as GDK_CONTROL_MASK and GDK_SHIFT_MASK into the mask. Usually you should just set the mask to be GDK_BUTTON1_MASK.

Note. The API documentation for this function states that the widget must have a window. I have used no-window widgets with success, and the source code for the function in GTK+-2.24 does not check whether the widget has a window.

You need to decide what types of data the widget will supply. Usually this is a simple matter; it has text, or perhaps images, or perhaps it has its own application-specific data chunks. The target table should be defined as described in Sections 2.3 and 2.4. We will use the following target table in the examples that follow.

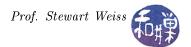
```
GtkTargetEntry target_entries[] = {
    {"text/html", 0, TEXT_HTML },
    {"STRING", 0, STRING},
    {"number", 0, NUMBER},
    {"image/jpeg", 0, IMAGE_JPEG},
    {"text/uri-list", 0, TEXT_URI}
};
```

The "text/uri-list" target is commonly used to drag links and filenames between applications.

You also need to connect the signals that you want to handle to the source widget.

4.1 Example

The following "boilerplate" listing demonstrates the basic steps in setting up a source widget.



```
void setup source widget
                                             ( GtkWidget *source widget )
    GtkTargetEntry target entries [] = {
         \{"text/html", 0, TEXT HTML \},\
                             0, STRING},
         {"STRING",
                           \begin{array}{c} 0 \ , \quad \text{NUMBER} \} \ , \\ 0 \ , \quad \text{IMAGE}_JPEG \} \ , \end{array}
         {"number",
         {"image/jpeg",
         {"text/uri-list", 0, TEXT URI}
    };
    /* Make this a drag source offering all of the targets listed above */
    gtk drag source set (source widget,
                          GDK BUTTON1 MASK,
                           target entries,
                           G N ELEMENTS (target entries),
                           GDK_ACTION_COPY | GDK_ACTION_MOVE);
    /* Connect it to all signals that the source should handle. */
    g_signal_connect(G_OBJECT(source_widget),
                       "drag begin",
                       G_CALLBACK (on_drag_begin),
                       NULL);
    g signal connect (G OBJECT (source widget),
                       "drag data get",
                       G CALLBACK (on drag data get),
                       NULL);
    g signal connect (G OBJECT (source widget),
                       "drag end",
                       G CALLBACK (on drag end),
                       NULL);
```

This widget will offer all of the different target types listed in target_entries. It also supports copying and moving of drag data. The only difference is that, on a move, the source has to delete the original data.

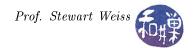
We will need to write the three handlers, on_drag_begin(), on_drag_data_get(), and on_drag_end(). These handlers depend upon the specific widget and what types of data it supplies. First let us see how to set up a destination widget, after which we will describe and give examples of handlers for all of the relevant signals.

5 Setting Up a Destination Widget

A widget is set up as a destination widget by calling gtk_drag_dest_set(), whose prototype is

```
void gtk_drag_dest_set ( GtkWidget *widget,
        GtkDestDefaults flags,
        const GtkTargetEntry *targets,
        gint n_targets,
        GdkDragAction actions);
```

The first argument, widget, is the widget to be the drag destination. The remaining arguments have the following meaning.



flags the default drag behaviors to use

targets the table of targets that the destination will accept, or NULL for none.

- n_targets the number of items in targets.
- actions the bitmask of possible actions for a drop onto this widget.

The **flags** argument can be used to specify default behaviors for this widget. The values make it possible to write a very simple destination widget if you are willing to accept default behaviors for it. If it is set to 0, there will be no defaults, and you will have to write handlers for all of the possible signals. The possible values for the **flags** argument as as follows:

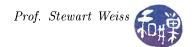
- GTK_DEST_DEFAULT_MOTION If set for a widget, GTK+, during a drag over this widget will check if the drag matches this widget's list of possible targets and actions, and will call gdk_drag_status() as appropriate. If you set this flag, you will not need to write a handler for the "drag-motion" signal; GTK+ will supply a default handler. Conversely, you should not set this flag if you do connect your own handler, because unless you really know what you are doing, there will be unpredictable results.
- GTK_DEST_DEFAULT_HIGHLIGHT If set for a widget, GTK+ will draw a highlight on this widget as long as a drag is over this widget and the widget drag format and action are acceptable. If you set this flag, then you will not need to do the highlighting yourself in the "drag-motion" handler, and if you do the highlighting there, then you should not set this flag.
- GTK_DEST_DEFAULT_DROP If set for a widget, when a drop occurs, GTK+ will check if the drag matches this widget's list of possible targets and actions. If so, GTK+ will call gtk_drag_get_data() on behalf of the widget and it will also gtk_drag_finish(). GTK+ will also take care of passing the appropriate values to gtk_drag_finish() to make sure that move actions and copy actions are handled correctly. If you set this flag, you will have to know what you are doing in your own custom "drag-motion" handler, and you will not need to write a handler for the "drag-drop" signal.
- GTK_DEST_DEFAULT_ALL If set, all of the above default actions will be taken. In this case, you will only need to write a handler for the "drag-data-receive" signal.

The destination does not have to accept the exact set of targets offered by the source. It might be a subset, or a superset, or it may even be unrelated, depending on the nature of your application. If DND is being used strictly to allow drags within your application, you may want to place the target table definition in a header file that can be included by all widgets, so that destination and source widgets share the same target names and info values.

5.1 Example

A listing of a set up of a very simple destination widget follows.

```
void setup_dest_button ( GtkWidget *dest_widget )
{
    /* Allow two different types of text */
    GtkTargetEntry text_targets[] = {
        {"text/html", 0, TEXT_HTML },
        {"STRING", 0, STRING}
    };
    gtk drag dest set( dest button,
```



```
0.
                    text_targets ,
                    0.
                    GDK ACTION COPY);
/* Connect this widget to all of the signals that a potential
   drop widget might emit. There are four of them: drag-motion,
   drag-drop, drag-data-received, and drag-leave.
*/
g signal connect (G OBJECT (dest widget),
                 "drag data received",
                 G CALLBACK (on drag data received),
                 NULL);
g signal connect(G OBJECT(dest widget),
                 "drag drop"
                 G CALLBACK (on drag drop),
                 NULL);
g signal connect (G OBJECT (dest widget),
                 "drag motion",
                 G CALLBACK (on drag motion),
                 NULL);
g signal connect (G OBJECT (dest widget),
                 "drag leave",
                 G CALLBACK (on drag leave),
                 NULL);
```

5.2 Adding Targets to the Destination

Although it is often sufficient to use gtk_drag_dest_set() to set up a destination widget, it is of limited use. It is often more convenient to create the destination with a NULL target table and then call gtk_drag_dest_set_target_list() to set the target list for the destination. The advantage of doing this is that there are several functions for adding classes of targets to target lists. For example, there is a function to add all image targets to a target list, or all text targets. The prototype of gtk_drag_dest_set_target_list() is

which is given the widget and a target list, not a target table. You can create a target list from a target table with

which is given the table and returns a newly-allocated target list. If you call it as

```
target_list = gtk_target_list_new(NULL, 0);
```

you will have an initially empty target list. You can prepend a target table to an existing target list with



void gtk_target_list_add_table

(GtkTargetList *list, const GtkTargetEntry *targets, guint ntargets);

and you can append a single target to a target list with

void	gtk_target_list_add	<pre>(GtkTargetList *list,</pre>
		GdkAtom target,
		guint flags,
		guint info);

This function requires the target's atom. This implies that you must have first interned the target with gdk_atom_intern() to get an atom for it. The flags value is the same as you would use in the target table, and the info value is the integer by which you want to refer to this target elsewhere in the application.

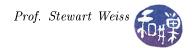
To add all image targets that are supported by the system's GtkSelectionData object, use

where info is the integer by which you will refer to all image targets, and writable is a flag that when TRUE, limits the image types to those that can be written, and when FALSE, allows all image types, those can can be read only as well as written.

You may refer to the API documentation for the other similar functions: gtk_target_list_add_text_targets(), gtk_target_list_add_uri_targets(), and gtk_target_list_add_rich_text_targets().

The order in which the targets occur in the target list is important. When the drop occurs, the widget will need to decide which of the targets to accept. The simplest function for this purpose will traverse the target list from front to back looking for the first target that satisfies the criteria. There are other methods of choosing that do not depend on order, but they will take more work.

The following listing shows how to set up a GtkDrawingArea widget to accept drops of any image format and the URIs. It might be interested in accepting the drop of a URI in case it is an image file, in which case it can load the image from the file.



```
NULL,
                             // empty target table
                   0,
                   GDK ACTION COPY);
/* Add the target list to the widget */
gtk drag dest set target list (drawing area, target list);
/* Initialize a pixbuf pointer to NULL to indicate there is no image */
g_object_set_data(G_OBJECT(drawing_area), "pixbuf", NULL);
/* Connect expose event handler */
g signal connect (G OBJECT (drawing area),
                 "expose-event",
                 G CALLBACK (on expose),
                 NULL);
/* Connect handlers for remaining signals */
g signal connect (G OBJECT (drawing area),
                 "drag data received",
                 G CALLBACK (on da drag data received),
                 NULL);
g\_signal\_connect(G\_OBJECT(drawing area),
                 "drag drop",
                 G CALLBACK (on da drag drop),
                 NULL);
g signal connect (G OBJECT (drawing area),
                 "drag motion",
                 G CALLBACK (on da drag motion),
                 NULL);
g signal connect (G OBJECT (drawing area),
                 "drag leave",
                 G CALLBACK (on da drag leave),
                 NULL);
```

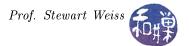
6 Signal Handlers in Depth

We will now show how to set up signal handlers for the source and destination widgets. To make things concrete, we will demonstrate this with a specific, but contrived example. We will create a single source widget: a button that can offer four types of target data: plain text, marked-up text, numbers, and an image. There will be four destination widgets: three buttons, one accepting numbers, one accepting plain text, and one accepting marked-up text, and a drawing area that can accept image data and text/uri-lists.

The widgets will all get packed into a top-level window so that the user can drag from the source onto any of the other widgets. Because the only data that a button has is its label, we will attach different types of data to the button as a GObject, so that it can supply data to destination widgets when they request it. The destination buttons will reflect their receipt of data by changing their labels to show the new data. The drawing area will display the images that it receives.

6.1 The Source Widget

The application's source button is set up with a call to setup_source_button(), shown in Listing 2. This gets the current time and save it in a local variable. It passes its value to the drag-begin handler. Most of the work is in the function make_drag_source(), shown in Listing 3.



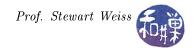
```
Listing 2: setup_source_button()
```

```
void setup source button
                                          ( GtkWidget *source button )
    time t start time;
    start time = time(NULL);
    /* Convert the button into a drag source */
    make drag source(source button);
    g signal connect (G OBJECT (source button),
                     "drag_begin",
                     G CALLBACK (on begin drag),
                     GUINT TO POINTER(start time));
    g signal connect(G OBJECT(source button),
                     "drag_data_get",
                     G CALLBACK (on drag data get),
                     NULL);
    g signal connect(G OBJECT(source button),
                    "drag end",
                     G CALLBACK (on end drag),
                     NULL);
```

The function make_drag_source() begins by defining the targets it will offer. The "number" target is limited to those widgets in the same application just to demonstrate what happens when it is dragged to another application when this flag is set. The function loads two pixbufs from hard-coded filenames and sets pointer to them as data in the button, so that it can deliver images when requested. The lines

set the target table and its length as data in the button object so that these will be available to other callback functions that need access to the offered targets (the drag-begin handler).

```
Listing 3: make drag source()
                             (GtkWidget *source button)
void make_drag_source
ł
    GError
                      * \operatorname{error} = \operatorname{NULL};
    GdkPixbuf
                      * pixbuf = NULL;
                      * pixbuf2 = NULL;
    GdkPixbuf
    /* Define the target data types that this widget can deliver */
    /* Even though this is declared on the stack, when the pointer is
       set on the object, it is reference counted and does not get deleted.
       I was surprised by this. */
    GtkTargetEntry target entries [] = {
         {"text/html",
                           0,
                                                    TEXT HTML \},
         {"STRING",
                            0,
                                                    STRING },
         {"number",
                            GTK TARGET SAME APP,
                                                   NUMBER },
                        0,
         {"image/jpeg",
                                                    IMAGE JPEG },
        {"text/uri-list", 0,
                                                    TEXT URI}
    };
```



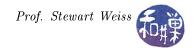
```
gtk drag source set (source button,
                     GDK BUTTON1 MASK,
                     target entries,
                     G N ELEMENTS (target entries),
                     GDK ACTION COPY | GDK ACTION MOVE);
/* Create a pixbuf from file data and error-check */
pixbuf = gdk_pixbuf_new_from_file ( BUTTON IMAGE PATH, &error );
if ( error != NULL)
{
    g print ("Failed to load image file: %s\n", error ->message);
    g error free(error);
    error = NULL;
}
/* Create a second pixbuf from file data and error-check */
pixbuf2 = gdk pixbuf new from file (BUTTON IMAGE2 PATH, & error );
if ( error != NULL)
{
    g print ("Failed to load image file: \%s \setminus n", error \rightarrow message);
    g_error_free(error);
    \operatorname{error} = \operatorname{NULL};
}
/* Attach the images to the widget so that it them to deliver */
g object set data (G OBJECT (source button),
                   "image", (gpointer) pixbuf);
g object set data (G OBJECT (source button),
                   "image2", (gpointer) pixbuf2);
/* Attach the targetlist pointer and the length of the array to the object
   so that the callbacks can reconstruct the array. */
g object set data (G OBJECT (source button), "targetlist",
                  (gpointer) target_entries);
g_object_set_data(G_OBJECT(source_button), "ntargets",
                   GUINT TO POINTER(G N ELEMENTS(target entries)));
```

The "drag-begin" signal handler must have the following prototype:

void	user_function	(GtkWidget	<pre>*widget,</pre>
			${\tt GdkDragContext}$	<pre>*drag_context,</pre>
			gpointer	user_data);

The drag_context is a pointer to a GdkDragContext structure. This structure will be passed to various callback functions during the drag operation. Both the source and the destination have a drag context structure. It contains

```
struct GdkDragContext {
   GObject parent_instance;
   GdkDragProtocol GSEAL (protocol);
   gboolean GSEAL (is_source);
   GdkWindow *GSEAL (source_window);
   GdkWindow *GSEAL (dest_window);
   GList *GSEAL (targets);
   GdkDragAction GSEAL (actions);
}
```



```
GdkDragAction GSEAL (suggested_action);
GdkDragAction GSEAL (action);
guint32 GSEAL (start_time);
};
```

Notice that it contains references to the source and destination windows, actions and suggested actions, a list of targets, the start time, and the protocol and parent. The details are not so important now. What matters is that you see that this drag_context is keeping track of the drag, and will be needed when it comes time to get targets and to determine actions.

The primary reason to have a handler for the "drag-begin" signal is to set up a custom drag icon. Our drag-begin handler will do a bit more than this. It will get the current time and compute the time elapsed between when the application started up and the current time. It will save this as the number to be offered to any widget asking for number data.

It will also construct more data based on the current time. It will create a plain text string stating the current time, e.g.:

It is now Sun Dec 11 20:57:46 2011

and it will create a Pango marked up string saying the same thing, but with formatting applied. It will create an array of four gpointers that will point to each of the four different data chunks that it can offer, indexed by the **info** members of the target array. In other words, the array is

target_data[TEXT_HTML] contains a pointer to the marked up text

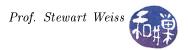
target_data[STRING] contains a pointer to the plain text

target_data[NUMBER] contains the number, cast to a gpointer

target_data[IMAGE_JPEG] contains a pointer to the pixbuf

The handler also creates the drag icon. When the user starts the drag, an image of the plain text string stating what time it is will be the icon. Because the code to create the icon is detailed and distracting, it is listed in the Appendix rather than here. The drag-begin handler is in Listing 4.

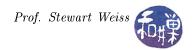
```
Listing 4: drag-begin handler
                                         ( GtkWidget
void on drag begin
                                                          *source button,
                                           GdkDragContext *dc,
                                                           user_data )
                                           gpointer
    GtkTargetEntry *target entries;
    gpointer *target data = NULL;
    guint
               start time = GPOINTER TO UINT(user data);
    GdkPixbuf *pixbuf
                           = NULL;
                            = NULL;
    gchar
              * html
    gchar
              *string
                           = NULL;
               elapsed time;
    guint
               which;
    guint
    size t
               length;
                                         /* Get the current time */
    time t
             now
                     = time(NULL);
    gchar
             *now string = ctime(&now); /* Convert it to a string */
    length = strlen(now_string);
    now string [length - 1] = ' \setminus 0';
                                         /* remove ending newline */
    /* get target entry data */
```



```
target entries = (GtkTargetEntry*) g object get data(
                                   G OBJECT(source button), "targetlist");
guint ntargets = GPOINTER TO UINT (g object get data(
                                   G OBJECT(source button), "ntargets"));
/* pick a random number using current time as seed */
srand(now);
which = rand();
/* Get the pixbuf image data pointer */
if (which \% 2 = 0)
    pixbuf = g object get data(G OBJECT(source button), "image");
else
    pixbuf = g object get data(G OBJECT(source button), "image2");
// Compute the time since the application started
elapsed time = now - start time;
// Create and draw the drag icon
draw drag icon(source button, dc, now string);
// Allocate the array of pointers for the target data, which is stored
// on the source button's object properties table
target_data = g_new(gpointer, ntargets);
g object set data (G OBJECT (source button), "target data",
                  target data);
// Define the content of the target data.
html = g markup printf escaped (
          "<span font family = \"monospace \" style = \"italic \">The time is "
          <span weight=| bold| foreground=| blue| >%s.</span>"
          || < | s pan > ||,
                                now string);
string = g strconcat("It is now ", now string, ".\n", NULL);
// Make a copy of the target data and store it in the array indexed so
// that we can retrieve it based on the TargetInfo enumerated type.
target data[TEXT HTML] = g strdup(html);
target data[STRING] = g strdup(string);
                      = GUINT TO_POINTER(elapsed_time);
target data [NUMBER]
if ( pixbuf != NULL ) {
    target data [IMAGE JPEG] = (gpointer) pixbuf;
}
/* We have to free the two strings, since we copied them into the target
   array, but cannot free the pixbuf, since we did not allocate it here,
   but just copied the pointer to a local. */
g free(html);
g free(string);
```

The most important signal handler for the source is the drag-data-get handler. When the drop is made on a destination widget, the "drag-data-get" signal is emitted on the source. The prototype for this handler is

void user_function		0	<pre>*widget, *drag_context, *data, info,</pre>
--------------------	--	---	---



guint time, gpointer user_data);

The data argument is a pointer to a GtkSelectionData structure that was initialized by GTK+ before emitting the signal. This structure has a member named target which is a pointer to the target requested by the destination. The info member is the number in the target list that corresponds to this target. The responsibility of the handler is to determine the target type and deliver the requested data. Therefore, its first task is to query the info member for the target data type. It can then copy data into the GtkSelectionData structure using the function

void	gtk_selection_data_set	<pre>(GtkSelectionData *selection_data,</pre>
		GdkAtom type,
		gint format,
		const guchar *data,
		<pre>gint length);</pre>

It needs the atom representing the target, the format (remember this is the number of bits in a unit of this type), data, a pointer to the data to be copied, and length, the number of units of this target type.

The exception is that pixbuf data is delivered in a different way. There is a special function that copies pixbuf data into a selection structure:

```
gboolean gtk_selection_data_set_pixbuf (GtkSelectionData *selection_data,
GdkPixbuf *pixbuf);
```

This returns TRUE if it was successful, and FALSE if it failed.

Our handler for the source button's "drag-data-get" handler is in Listing 5. The drag context is not used in this function. The important part of the code is the switch statement. It checks the type of data requested and in each case, copies the data into the selection structure. Although we use instructions like

```
gdk_atom_intern("text/html", TRUE);
```

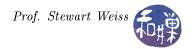
to get the atom for the target, it would be simpler to replace these by

```
selection_data->target
```

which is the requested atom. This code just shows another way to do the same thing.

Listing 5: drag-data-get handler for source button

void on_drag_data_get	(GtkWidget	$*$ source _ button ,
	${ m GdkDragContext}$	*dc,
	GtkSelectionDa	ta *selection data,
	guint	info,
	guint	time,
	gpointer	user data)
{	01	_ ,
guint number data;		
GdkPixbuf *pixbuf = NULL;		
gpointer *target data =		
g_object_get_data(G_OBJE	CT(source button)	"target data"):
8_0.1001_801_aata (a_0101	er(source_succon);	taiset_aata),

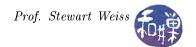


```
/* Use the info argument to determine which data the destination has
  requested, and set up the GtkSelectionData structure to be passed to
  the destination through the drag-data-received callback.
  The target data array was filled in such a way that the info value
  equals the array index, so we set the data for info value X from
  target data[X].
 The exception is the image data. There are separate functions for
  supplying pixbuf, uri, and utf-8 data.
*/
switch(info)
{
    case TEXT HTML :
        gtk selection data set (selection data,
            /* We could use
               selection data->target
               instead of getting the interned atom each time
            */
            gdk atom intern("text/html", TRUE),/* type
                                                           */
                                                /* format */
            8,
            target_data[TEXT HTML],
                                                /* data */
            strlen(target data[TEXT_HTML]));
                                                /* length */
        break;
    case STRING :
        gtk selection data set (selection data,
            gdk atom intern("STRING", TRUE), /* type
                                                        */
                                              /* format */
            8,
            target data[STRING],
                                              /* data */
                                              /* length */
            strlen(target data[STRING]));
        break;
    case NUMBER :
        number data = GPOINTER TO UINT(target data[NUMBER]);
        gtk selection data set (selection data,
            gdk_atom_intern("number", TRUE), /* type
                                                         */
            32,
                                               /* format */
            (guchar*) &number data,
                                               /* data
                                                         */
                                               /* length */
            1);
        break;
    case IMAGE JPEG :
        pixbuf = target data[IMAGE JPEG];
        gtk selection data set pixbuf(selection data, pixbuf);
        break;
}
```

The source widget has two remaining signals that it should handle. One is "drag-end" and the other is "drag-delete". In our case there is nothing to do for the drag-delete handler. If we wanted to do a real move instead of a copy, then we would have to delete data in the source widget when the drag finished, but in this example we do not do this.

However, the "drag-end" signal should be handled because we need to de-allocate the strings that we created with the current time. We no longer need them. The prototype for the drag-end handler is

void	user_function	(GtkWidget *widget,
		GdkDragContext *drag_context,



gpointer user_data);

Our handler does not use the drag context. It does not need to free any resources used by the drag icon because GTK+ takes care of unref-ing the icon's pixmap when the drag finishes. Our handler is in Listing 6.

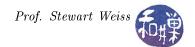
Listing 6: drag	-end handler
void on_end_drag	(GtkWidget *source_button, GdkDragContext *dc, gpointer user_data)
// Retrieve the array of pointers to gpointer *target_data = g_object_get_data(G_OBJECT	<pre>source supplied data (source_button), "target_data");</pre>
<pre>// Release the memory that was alloca // drag source g_free(target_data[TEXT_HTML]); g_free(target_data[STRING]); g_free(target_data);</pre>	ated when widget was set as a
// Set the key to NULL g_object_set_data(G_OBJECT(source_bu	<pre>utton), "target_data", NULL);</pre>

6.2 The Button Destination Widgets

There are four destination widgets: three buttons and one drawing area. The buttons all share the same functions. To make this possible, each button will have an ID number attached to it as user data, and this ID will tell the callbacks which button is the destination widget. There are also more signals to be handled on the destination side.

The code to set up the destination buttons is in Listing 7. The button's id is passed as user data to the setup function, and it is then passed as user data to each of the callback functions. Separate target lists are created for the number and text buttons.

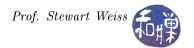
Listing 7: setup_dest_button()



```
gtk drag dest set (dest button,
                    0,
                    NULL,
                    0.
                    GDK ACTION COPY);
switch (button type)
ł
case NUMBER DROP:
    target list = gtk target list new(number targets,
                               G N ELEMENTS(number targets));
    break;
case MARKUP DROP:
case TEXT PLAIN DROP:
    target list = gtk target list new(text targets,
                               G N ELEMENTS(text targets));
    break;
default:
    g_print ("Unidentified button.n");
}
/* Now add the target list created above to the widget */
gtk drag dest set target list(dest button, target list);
/* Connect this widget to all of the signals */
g signal connect (G OBJECT (dest button),
                 "drag data received",
                 G CALLBACK (on db drag data received),
                 GUINT TO POINTER(button type));
g signal connect (G OBJECT (dest button),
                 "drag drop",
                 G CALLBACK (on db drag drop),
                 GUINT TO POINTER(button type));
g signal connect (G OBJECT (dest button),
                 "drag motion",
                 G CALLBACK (on db drag motion),
                 GUINT TO POINTER(button type));
g signal connect (G OBJECT (dest button),
                 "drag leave",
                 G CALLBACK (on db drag leave),
                 GUINT TO POINTER(button type));
```

The first handler that we need to create is the "drag-motion" signal handler. This signal is emitted on a destination when the cursor moves over it during a drag. The callback prototype is

gboolean	user_function	<pre>(GtkWidget *widget, GdkDragContext *drag_context,</pre>
		gint x,
		gint y,
		guint time,



gpointer user_data);

The two arguments of interest here besides the drag_context are the x and y coordinates. These are the cursor's position in the coordinates of the widgets GDK window. The handler may use these to determine whether a drop is possible in that particular place, or if there are different types of data that can be dropped depending on the position.

This handler has several roles. One is to determine whether it can accept a drop. Another is to highlight the widget to give the user visual feedback that a drop is possible. To do either of these things, it needs to check the targets that are being offered by the drag and compare them to the targets in its list of acceptable targets. The function

GdkAtom	gtk_drag_dest_find_target	(GtkWidget *widget,
		GdkDragContext *context,
		<pre>GtkTargetList *target_list);</pre>

can be used for this purpose. It is given the destination widget and a pointer to the passed in drag context. The third argument is a target list. The function will compare the targets stored in the drag context against those in the target list, traversing the two lists from front to back, and stopping when it finds the first target that the source is offering that is also in the target list.

The proper way to use this function is to pass NULL as the third argument. By doing this, GTK will use the list returned by the call gtk_drag_dest_get_target_list(widget), which is the list of targets that the destination widget can accept.

If a target is found, its atom is returned, otherwise GDK_NONE is returned. If no target was found, the drag motion handler needs to notify the source that it will not accept a drop. It does this by calling

void	gdk_drag_status	<pre>(GdkDragContext *context,</pre>
		GdkDragAction action,
		<pre>guint32 time_);</pre>

To indicate that a drop will not be accepted, it passes a 0 as the second argument and returns FALSE. It can pass the time value argument given to the drag motion handler as the third argument.

On the other hand, if an atom was found, then the handler should highlight the widget and call gdk_drag_status() to indicate the action that should be taken when the drop happens. This can be any of the values

GDK_ACTION_DEFAULT Means nothing, and should not be used.

- GDK_ACTION_COPY Copy the data.
- GDK_ACTION_MOVE Move the data, i.e. first copy it, then delete it from the source using the DELETE target of the X selection protocol.
- GDK_ACTION_LINK Add a link to the data. Note that this is only useful if source and destination agree on what it means.
- GDK_ACTION_PRIVATE Special action which tells the source that the destination will do something that the source doesn't understand.

GDK_ACTION_ASK Ask the user what to do with the data.

Rather than making this decision itself, the handler can simply call

gdk_drag_status(context, context->suggested_action, time);

gtk_drag_highlight



which will tell GTK+ to use the suggested action as the action to take. GTK+ will base this on default settings, such as whether the control key is pressed. Lastly, to highlight the widget, it should call

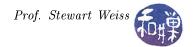
void

(GtkWidget *widget);

which will highlight the widget and attach expose event and draw handlers to it. But if you do this, then you also have to handle the "drag-leave" signal, because if you highlight the widget when the cursor is over it, then you have to unhighlight it when the cursor leaves it. This requires keeping some state around to know what its state of highlighting is. Our example drag-motion and drag-leave handlers are in Listing 8. The drag-motion handler has to return TRUE if a drop is possible.

T''' O	1	1 1 1	1 11
LISTING X	drag-motion	and drag-le	ave handlers
Libung O.	ulag motion	and drag it	ave handlers

```
( GtkWidget
gboolean on db drag motion
                                                    *dest button,
                                         GdkDragContext *dc,
                                         gint
                                                       х,
                                         gint
                                                       у,
                                         guint
                                                       time,
                                                       user data )
                                         gpointer
   GdkAtom target atom;
   static int counter = 0;
   target atom = gtk drag dest find target(dest button, dc, NULL);
   if ( target atom != GDK NONE)
   {
       gdk drag status(dc, dc->suggested action, time);
       // If the destination widget isn't highlighted yet and an action
       // has been suggested by GTK, higlight this destination widget.
       if ( !g object get data(G OBJECT(dest button),
                               "highlighted") && dc->suggested action)
       {
           g object set data(G OBJECT(dest button),
               "highlighted", (gpointer)TRUE);
           gtk drag highlight (dest button);
       }
       /* The widget is a valid destination. */
       return TRUE;
   }
   gdk drag status(dc, 0, time);
   return FALSE;
void on db drag leave
                                                      *dest button,
                                       ( GtkWidget
                                        GdkDragContext *dc,
                                        guint
                                                       time,
                                                       user data )
                                        gpointer
   gtk drag unhighlight(dest button);
   g object set data(G OBJECT(dest button), "highlighted", (gpointer)FALSE);
```



The destination has to have a handler for the "drag-drop" signal, which occurs when the user releases the mouse button over the widget. The prototype for this handler is

gboolean user_function	<pre>(GtkWidget *widget, GdkDragContext *drag_context, gint x, gint y, guint time, gpointer user_data);</pre>
------------------------	--

which is the same as that of the drag motion handler, and the arguments have the same meanings. The drag-drop handler has to request the data from the source widget. To do this it needs to know which data to request. If there were no drag-motion handler, this handler would have to check the targets being offered against those in the destination's target list, just as was done in the drag motion handler. But since that work was already done, we know that a drop is possible, and all that a handler has to do is to ask for the data by calling

void	gtk_drag_get_data	(GtkWidget *widget,
		GdkDragContext *context,
		GdkAtom target,
		<pre>guint32 time_);</pre>

passing the required target atom. How does it know which target to request? It could look in its target list. In our case, each destination button requests a specific type of data, so we just inspect the user data, which encodes the button's ID, and request that target that the button is designed to accept. Listing 9 clarifies this.

Listing 9: drag-drop handler						
gboolean on_db_drag_drop (GtkWidget GdkDragContext gint gunt gpointer	*dc, x, y, time,				
$GdkAtom target_atom = GDK_NONE;$						
// Decide which widget emitted the signal, based on user_data DestinationType button_type = GPOINTER_TO_UINT(user_data);						
<pre>// The target_atom to request depends on the widget. The Markup button will // ask for the text as markup; the Number button will ask for the data as // a number. The plain text choice has no widget at present. if (button_type == MARKUP_DROP) </pre>						
<pre>target_atom = gdk_atom_intern("text/html", TRUE); else if (button_type == TEXT_PLAIN_DROP) target_atom = gdk_atom_intern("STRING", TRUE);</pre>						
else if (button_type == NUMBER_DROP) target_atom = gdk_atom_intern("number", TRUE);						
// This should always be true, unless the atoms got deleted somehow. if (target atom $!=$ GDK NONE)						



```
gtk_drag_get_data(dest_button, dc, target_atom, time);
```

return TRUE;

The destination widget then needs the handler for the "drag-data-received" signal, which is emitted on it when the source has supplied the data. The callback prototype is

void	user_function	<pre>(GtkWidget *widget, GdkDragContext *drag_context, gint x, gint y, GtkSelectionData *data, guint info</pre>
		guint info,
		guint time,
		<pre>gpointer user_data);</pre>

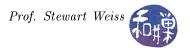
The parameters are at this point self-explanatory, except possibly for the (x,y) coordinates. This is the position where the drop occurred.

The responsibility of this handler is to get the data from the GtkSelectionData structure and copy it into its own variables, and to call gtk_drag_finish() to notify the source that the drag has been completed. In the simple case, it does not need to look at the position of the drop. It should look at the value of drag_context->action to see if it is a move or a copy, or whether it is GDK_DRAG_ASK, in which case it should display a dialog box asking the user what to do.

In our application we just treat the action as a copy and forgo checking the action. Again, because the callback has to do different things for different buttons, its logic is based on the value of the button id passed as user data. It is in Listing 10.

Listing 10: drag-data-received handler

void	on_db_drag_data_received	(GtkWidget GdkDragContext gint GtkSelectionDat guint guint gpointer	*dc, x, y, a *selection_data, info, time,		
1						
	$\operatorname{gchar} * \operatorname{label} = \operatorname{NULL};$					
	guint elapsed time;					
	DestinationType button_type = GPOINTER_TO_UINT(user_data); switch (button_type) { case MARKUP DROP:					
	label = g strconcat((gchar*)s)	sel	ection data->da	ata, "", NULL):		
	gtk label set markup (GTK LABEL(GTK BIN(dest button)->child), label);					
	$g_free(label);$					
	break;					
	case TEXT_PLAIN_DROP:					
	label = g strconcat((gchar*)selection data->data, "", NULL);					
	gtk label set text (GTK LABEL (GTK BIN (dest button) -> child), label);					
	g free(label);					
	break;					

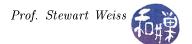


```
case NUMBER_DROP:
    elapsed_time = (guint)*(selection_data->data);
    label = g_strdup_printf ("The drag started %u seconds after"
                                 " the application started.\n", elapsed_time);
    gtk_label_set_text(GTK_LABEL(GTK_BIN(dest_button)->child), label);
    g_free(label);
    break;
default:
    g_print(" Got some other data.\n");
}
// Indicate that the drag is finished and free the target_name string
gtk_drag_finish( dc, TRUE, FALSE, time);
```

6.3 The Drawing Area Widget

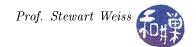
This widget is also a destination widget. Its setup was displayed in Listing 1. The signal handlers for it are conceptually the same as those for the buttons, but the technical details are different. We describe here how to extract image and text/uri-list data in the "drag-data-received" handler, and how to check for valid drop targets in the motion and drop handlers.

(to be continued...)



Appendix

Listing 11: Drawing a Custom Drag Icon void draw drag icon (GtkWidget *widget, GdkDragContext *drag context, gchar *text) GtkStyle * style = NULL;GdkPixmap *icon pixmap = NULL;GdkPixmap *icon mask = NULL;*pixmap_gc = NULL; GdkGC GdkGC $*mask_gc = NULL;$ mask color; GdkColor PangoFontDescription *fd; PangoLayout *layout; PangoContext *context; width, gint height; // Get the style of the widget. To be sure it actually has a style, we // call gtk_widget_ensure_style, which basically tests that the widget is // realized. gtk widget ensure style(widget); style = gtk widget get style(widget); /*** The next few steps are required to draw text into a pixmap. ***/ // Get the current font description from the widget's GtkStyle $fd = widget \rightarrow style \rightarrow font desc;$ // Get a PangoContext with the appropriate font map, font description, // and base direction for this widget. context = gtk widget get pango context(widget); // Create a new PangoLayout object with attributes initialized to // default values for the newly created PangoContext. layout = pango layout new (context); // We have to determine the size of the rectangle that will be used to // render the text. We set the font description in the layout and then // copy the text into it. The layout can then calculate the width and height // of the rendered text, measured in PANGO distance units !! pango_layout_set_font_description (layout, fd); $\verb"pango_layout_set_text" (layout, text, -1);$ pango_layout_get_size (layout, &width, &height); // PANGO_SCALE is the number of pango units per pixel, so we scale the // width and height into pixels width = width / PANGO SCALE; height = height / PANGO SCALE; // Create and allocate the icon pixmap and mask $icon_pixmap = gdk_pixmap_new(widget \rightarrow window, width, height, -1);$ icon mask = gdk pixmap new(NULL, width, height, 1); // Next we initialize the icon pixmap and mask graphical contexts pixmap gc = gdk gc new(icon pixmap);gdk_gc_set_background(pixmap_gc, &style ->white);



```
gdk gc set foreground(pixmap_gc, &style->black);
gdk gc set function(pixmap gc, GDK COPY);
mask\_gc = gdk\_gc\_new(icon\_mask);
mask color.pixel = 1;
gdk gc set foreground (mask gc, & mask color);
gdk gc set function(mask gc, GDK COPY);
//\ Now we draw a white background for the icon rectangle and set the
// foreground color of the gc to black
gdk_gc_set_foreground(pixmap_gc, &style ->white);
gdk\_draw\_rectangle(icon\_pixmap, pixmap\_gc, TRUE, 0, 0, width, height);
gdk\_gc\_set\_foreground(pixmap\_gc, \&style \rightarrow black);
// Make the icon completely opaque
gdk_draw_rectangle(icon_mask, mask_gc, TRUE, 0, 0, width, height);
mask color.pixel = 0;
gdk gc set foreground(mask gc, &mask color);
// Render the text into the icon pixmap, cast as a GtkDrawable
gdk draw layout (GDK DRAWABLE(icon pixmap),
                 pixmap\_gc\,,
                 0,
                 0,
                 layout);
// Set the icon as the drag icon.
gtk\_drag\_set\_icon\_pixmap(drag\_context,
    gdk colormap get system(), icon pixmap, icon mask,
    width /2, height /2;
/* unref the layout */
g object unref (layout);
```