18
Selections

The X Window System provides several methods for separate applications to exchange information with one another. One of these methods is the use of selections. A *selection* transfers arbitrary information between two clients. An in-depth discussion of the selection mechanism that X provides is available in the *Xlib Programming Manual*. This chapter describes XView's new selection package. Previous revisions of XView provided a selection mechanism that did not use objects.

The selection mechanism for previous versions of XView used special functions and structures to implement selections. The older selection mechanism is still supported and it is described in Appendix A, *The Selection Service*. The new package implements selections as objects. The application programmer interface for the new selection package conforms to the standard XView model, using `xv_create()`, `xv_set()`, and `xv_get()`.

XView selections are used to exchange data between different applications or for communications within a single application. Many, but not all selections use text; a section of text is selected, then the selected text is cut or copied, and pasted to another area. The selection mechanism is not limited to text. For example, selections can be used to transfer filenames, sound data, a file's timestamp, or other information. Selections require only that the sender and the recipient have knowledge of the format of the data being transferred. Therefore, selections can be used to transfer graphics between applications that can understand a common format for communicating graphics. XView selections conform to the conventions in the ICCCM (Interclient Communications Conventions Manual). For a description of the ICCCM conventions, refer to Volume Zero, *X Protocol Reference Manual*.

In OPEN LOOK, you select objects in basically the same way that you select windows or icons —using the SELECT and ADJUST mouse buttons. Figure 18-1, from the *OPEN LOOK GUI Specification Guide*, shows one way to select text. OPEN LOOK describes three functions that operate on selected objects: CUT, COPY, and PASTE. These are *core functions* that are accessed from the keyboard.* Table 18-1 summarizes text selection for the OPEN LOOK GUI.

*The function keys that are bound to these functions vary from keyboard to keyboard depending on the make and model of the computer.

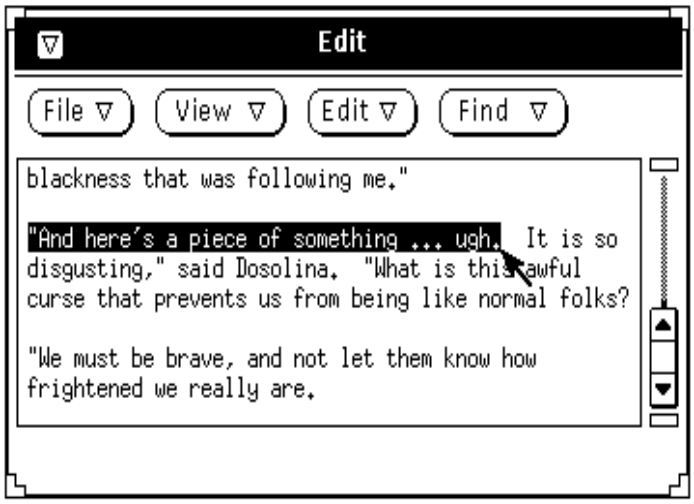

Figure 18-1. Dragging the pointer to select text

Table 18-1. Selecting Text

Action	Off Selection	On Selection
Click SELECT	Insert point is set at the pointer location.	When SELECT is released, insert point is set at pointer location and selection is cleared.
Drag SELECT	Text is highlighted as pointer is dragged (wipe-through selection).	Text move pointer is displayed. When you release SELECT, text is moved to pointer location if that location is outside the highlighted area.
Click ADJUST	Extends the highlighting to the pointer location extending either the beginning or the end of the current selection.	Moves the end of the highlighting to the pointer location. Beginning of selection is preserved.
Drag ADJUST	Adjusts an existing selection as the pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.	Adjusts an existing selection as pointer is dragged (wipe-through). Beginning of selection is anchored at insert point.

If an application uses objects such as `TEXTSW` or `PANEL_TEXT_ITEM`, the CUT, COPY, and PASTE command keys are internally implemented. These selection functions are built-in for these objects.

Note that although OPEN LOOK specifies the selection of graphic objects, no XView objects currently support selection of graphical objects. A possible implementation is to have a canvas object, which has graphic objects displayed in it, set selections based on the event

sequences outlined in Table 18-1. A *draw* application might consider a drawn geometric shape as a graphic object, whereas a *paint* application might consider the pixels in an arbitrary area as the graphical object.

18.1 The XView Selection Model

The XView selection model is based upon the requestor/owner model of peer-to-peer communications. Selections communicate data between an *owner client* and a *requestor client*. An owner client has the selection data and the requestor client wants that data. XView uses an object instantiated from the `SELECTION_OWNER` package to handle the selection owner, and an object instantiated from the `SELECTION_REQUESTOR` package to handle the selection requestor. Besides the owner and requestor objects, there is an XView hidden class, `SELECTION`, that handles information common to both a requestor and an owner. XView also provides an *optional* selection-item object, instantiated from the `SELECTION_ITEM` package, that can simplify the owner side of selections (to simplify this discussion, we do not cover the selection item until later in this chapter).

For XView to implement selections, XView tracks all X events that are generated from selections. These events are: `SEL_CLEAR`, `SEL_NOTIFY`, and `SEL_REQUEST` (X selection events are mapped to these XView events). Your application should ignore these events if you are using the selection package. If you want to implement selections with Xlib routines, then you can use these events; however, mixing is not allowed. That is, do not try to use both these events *and* use the XView the selection package.

The selection package allows you to select the *rank* of your selection. The rank is an atom representing a name on the server. The default rank is called the primary rank (`XA_PRIMARY`). For selections involving text, the *primary* rank is normally indicated on the screen by inverting (*highlighting*) its contents. Selections made while a function key is held down are considered *secondary* selections* (for text selections this is usually indicated with an underscore under the selection).

Other ranks are also available; OPEN LOOK uses the CLIPBOARD rank to hold data that has been cut or copied, using the cut or copy operations (with the CUT or COPY keys). This selection is inserted into an application with the paste operation (using the PASTE key). Atoms take up server resources, therefore the primary, secondary, and clipboard ranks are commonly used for selections; they are used over and over unless more than three selections need to be created. Other selections are used when the primary selection must be left undisturbed.

When a user interface element, such as a text subwindow wants to allow the user to make a selection, internally, it creates a *selection-owner* object. Through this selection-owner object, the text subwindow can *acquire* a selection rank (primary, secondary, etc) and provide it with data. An application can have many such selection owners, but there is typically one selection owner per XView object. Normally a selection rank is associated with an owner—a

*Which function key depends on your particular computer. By default the `L6` function key should work for Sun Workstations or the `F6` key for other computers.

separate owner need not be created just to utilize other selection ranks. However, only one owner can acquire a particular selection rank at any one time. If one application acquires a selection, and then another application acquires the same selection, the first application loses the selection ownership. The application that becomes the new selection owner may change the data associated with the selection.

Since the selection package conforms to the conventions of the ICCCM, selections between non-XView applications should be seamless.

18.2 How Selection Works (Without a Selection Item)

Here is a brief overview of the steps that take place during a single transfer of data from one XView application to another. We'll assume we have two applications, application *A* and application *B*, either of which can operate as the owner or the requestor.

Initially, both applications are in exactly the same state, neither has acquired any selections, and neither is the owner or the requestor. We'll also assume that selections are implemented using actions tied to mouse buttons or the CUT and PASTE function keys.

1. The user selects an item in application *A* and the application highlights the item. At this time, application *A* would acquire the primary selection.
2. In the XView Selection Owner package, nothing further happens until an `ACTION_COPY` event is delivered. At this time, application *A* acquires the selection and the primary selection is copied to the clipboard.
3. The user pastes the selected data into application *B*, causing an `ACTION_PASTE` to be sent to application *B*.
4. The `ACTION_PASTE` event may cause the selection requestor to request to receive the selected data into application *B*.
5. The selection data request invokes an application-defined conversion procedure which is associated with the Selection Owner in application *A*. It is the responsibility of the Selection Owner to convert the data into a format specified by application *B* or to reject the request.
6. Once the data conversion is finished, successfully or unsuccessfully, an application-defined reply procedure is invoked in application *B*. This procedure is associated with the selection requestor and either displays the data in application *B*, if the data conversion was successful, or indicates that the kind of data selected in application *A* cannot be pasted in application *B*, or that the kind of data requested by *B* cannot be supplied by *A*.
7. Once the selection data transfer is completed, a application-defined done procedure may be called by the selection owner. The done procedure may be used to free the memory associated with the selection, or perform other cleanup that is required.
8. If another application acquires the selection, application *A*'s "lose" procedure is called. This procedure is used to tell the selection owner (application *A*) that it has lost

ownership of the selection. For example, the lose procedure might unhighlight text that was previously selected and highlighted.

18.2.1 Highlighting the Selection (Selection Owner)

The first task for the application that is to become the selection owner is to mark the selection. For example, in a selection transfer involving text, when the pointer is placed over the text, an `ACTION_SELECT` should highlight the selection. While `LOC_DRAG` occurs, the highlight would be extended over the selection. If you are creating your own selections and you want your application to be OPEN LOOK compliant, you need to handle events and highlighting as specified in the *OPEN LOOK GUI Functional Specification*. Some XView packages, such as `TEXTSW` and `PANEL_TEXT_ITEM`, provide this functionality internally.

18.2.2 Making the Selection (Selection Owner)

After the user selects an item and it is highlighted, the application waits to receive an event of interest. For example, in a text subwindow the act of highlighting an item causes the primary selection to be acquired. At this time the XView application creates a selection owner object, if one was not previously created. For the text subwindow example, an `ACTION_COPY` event causes the CLIPBOARD selection to be acquired (note that in this case two selections are acquired since the COPY operation in OPEN LOOK uses the CLIPBOARD).

Applications perform three steps to become the selection owner:

1. A `Selection_owner` object is created using `xv_create()` (assuming a previously created selection owner is *not* being reused).
2. A mechanism for replying to the selection request should be set. This may be accomplished either by setting a conversion procedure or by creating a selection item (see Section 18.3.1, “The Selection Item”). The conversion procedure, or the selection item, allows the selection data to be associated with the selection owner and converted to the type a requestor expects.
3. The selection must be acquired.

A selection-owner object is created using `xv_create()`:

```
Selection_owner   sel_owner;
sel_owner = xv_create(window, SELECTION_OWNER,
                      NULL);
```

A selection-owner object’s owner is a window or a window based object. The `SELECTION_OWNER` package is defined in the header file *<xview/sel_pkg.h>*. Programs that use a selection owner must include this file. Figure 18-2 shows the class hierarchy for the selection-owner object.

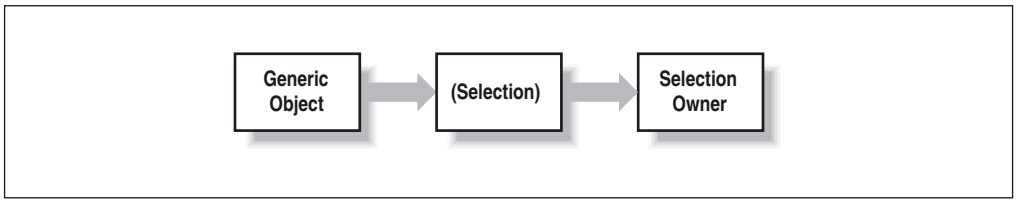

Figure 18-2. Selection owner class hierarchy

If a selection uses a conversion procedure to convert the selection, then the attribute SEL_CONVERT_PROC specifies the name of the conversion procedure. Section 18.2.4, "Converting the Selection," describes the conversion procedure. The selection is acquired by setting SEL_OWN to TRUE.

```
xv_set(sel_owner, SEL_CONVERT_PROC, convert_proc,
                  SEL_OWN, TRUE,
                  NULL);
```

By default, the selection is associated with the primary rank (XA_PRIMARY). If your selection needs to use another rank, use either SEL_RANK or SEL_RANK_NAME to specify the rank.

```
xv_set(sel_owner, SEL_CONVERT_PROC, convert_proc,
                  SEL_RANK_NAME, "SECONDARY",
                  SEL_OWN, TRUE,
                  NULL);
```

SEL_RANK takes an atom as an argument. SEL_RANK_NAME takes a string as an argument and creates an atom from the string; if the atom already exists, it is re-used. In X terms, this is called interning an atom. (See the description for XInternAtom in the *Xlib Reference Manual*.)

After an application makes a request and a selection owner responds to that request, the selection owner waits for notification that the requestor received the data. The maximum time that the selection owner will wait for an acknowledgement from the selection requestor is set with SEL_TIMEOUT_VALUE. This value is specified in seconds. If this value is exceeded, the selection is invalid. This value does *not* limit how long a selection owner may hold a selection. A selection owner may hold a selection for any length of time, unless another owner acquires the same selection, in which case the owner loses the selection ownership.

Note that the attributes SEL_RANK, SEL_RANK_NAME, and SEL_TIMEOUT_VALUE are SELECTION attributes. They apply to both the selection owner and to the selection requestor.

When and if a selection owner receives a selection request, the selection data needs to be converted. To convert a selection an application can create a selection-item object, or it can define a selection conversion procedure. We cover these possibilities in later sections.

18.2.3 Requesting the Selection (Selection Requestor)

When the user pastes the selection into an application, the ACTION_PASTE event should be used to trigger the application to become the selection requestor. A selection requestor requests to receive the selected data. The selection requestor allows an application to obtain selection data in a specified form. XView permits the selection requestor to receive a selection in one of two ways:

- A blocking request. The application is blocked until the transfer completes, either successfully or unsuccessfully.
- A non-blocking request. The application is not blocked during the selection data transfer.

This section covers both blocking and non-blocking requests.

An application requests a selection from a selection owner by performing two steps: first, the application creates a selection requestor object and sets its attributes. Second, the request is "posted." Posting the request sends a request to the selection owner.

Create a selection requestor from the SELECTION_REQUESTOR package:

```
Selection_requestor  sel_requestor;
sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                          NULL);
```

The SELECTION_REQUESTOR package is defined in the header file *<xview/sel_pkg.h>*. The owner of a selection requestor object is a window-based object. The class hierarchy for a selection requestor is shown in Figure 18-3.

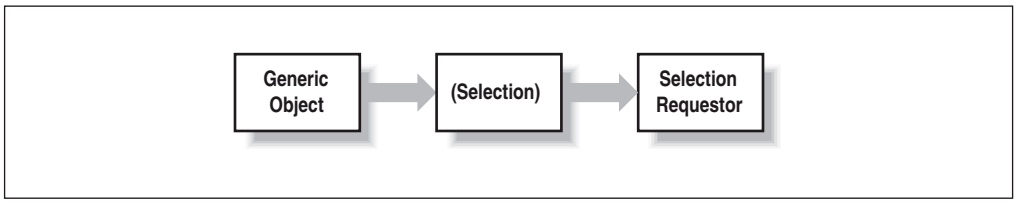

Figure 18-3. Selection requestor class hierarchy

The primary rank is the default rank used in a selection request. To use a rank other than the primary, use either SEL_RANK or SEL_RANK_NAME to specify the desired rank.

The time of the event that triggered selection request, the ACTION_PASTE event, should be set using the attribute SEL_TIME. If you do not set this attribute, the selection package does its best to set the SEL_TIME internally; however, this requires additional processing that is not required if the application sets SEL_TIME.*

*SEL_TIME fixes certain race conditions that would occur if the time of the events associated with selection the were not known.

18.2.3.1 Specifying the target type (selection requestor)

By default, a selection requestor uses the primary rank with the target set to string, which asks the selection owner to convert string data. When the requestor wants the owner to send data, it informs the owner of the type of information it expects using a selection *target*. The target may represent the type of the selection data; however, the selection transfer may not actually involve the selection data. A selection requestor may simply request a timestamp from the selection owner, or some other characteristic of the selection data, such as its length. The selection requestor can also request that the selection owner send a list of the types it can convert. When the requestor specifies the target named `TARGETS`, the owner should return a list of the types that it is able to convert to.

The attribute `SEL_TYPE` specifies the atom name for the requested target. The default value is `XA_STRING`. This may be set to `XA_INTEGER`, or to another atom. `SEL_TYPE_NAME` uses a string argument for the type, such as "`INTEGER`," internally, the package converts this string to an atom.

A single selection request may be used to request more than one target. For example, the selection requestor may desire a list of the possible types the owner can convert as well as selection data. This type of a request, where more than one target is requested at a time, is a *MULTIPLE* request. To initiate a multiple request, use the attribute `SEL_TYPES` with a `NULL`-terminated list of atoms to specify the targets requested, or use `SEL_TYPE_NAMES` with a `NULL`-terminated list of strings.

Two additional attributes allow the requestor to make a multiple request. `SEL_APPEND_TYPES` appends a `NULL`-terminated list of target types on to the existing list. `SEL_APPEND_TYPE_NAMES` appends a `NULL`-terminated list of names on to the current list. By default, a selection requestor selects a string, so the following request adds `TARGETS` to the request list, thus creating a multiple request:

```
xv_set(sel, SEL_APPEND_TYPE_NAMES, "TARGETS", NULL, NULL);
```

18.2.3.2 SEL_REPLY_PROC (selection requestor)

The selection requestor uses `SEL_REPLY_PROC` to specify the reply procedure. This attribute takes a function pointer as an argument. The reply procedure is used as a communication mechanism between the selection owner and the selection requestor. The reply procedure is invoked by the selection package after the conversion, when the response from the selection owner arrives.

If the request is a multiple request, the reply procedure is called once for each target requested:

```
Selection_requestor  sel_requestor;

sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                  SEL_REPLY_PROC, ReplyProc,
                                  NULL);
```

The reply procedure is described in Section 18.2.5, "Handling the Response."

18.2.3.3 Timeout for a selection response

The attribute SEL_TIMEOUT_VALUE specifies the maximum time that the selection requestor expects the particular request conversion to take. This should be set to the time selection owner will take to convert the selection and return data to the selection requestor.

18.2.3.4 Requesting the CLIPBOARD selection—blocking

To make a blocking request, the selection requestor uses the SEL_DATA attribute to initiate the request. Note that a blocking request puts the entire application on hold during the selection transfer. This type of selection transfer is not recommended for large selections, since the application may be put on hold for a significant amount of time. SEL_DATA takes two arguments, a pointer to an `unsigned long` which is set to the number of elements returned in the selection buffer, and a pointer to an integer which is set to the data format. Using `xv_get()` with the selection requestor and SEL_DATA, with the specified arguments returns the selection data. Example 18-1 shows how the selection requestor requests the clipboard selection using a blocking request.

Example 18-1. Requesting the CLIPBOARD selection--blocking

```
Selection_requestor sel_requestor;

main()
{
          .
          .
          .
     sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                SEL_RANK_NAME "CLIPBOARD", NULL);
          .
          .
          .
}

window_event_handler(..., event, ...)

Event          *event;

{
    char           *data;
    int            format; /* size of data element: 8,16, 32 bits.*/
    unsigned long  length; /* number of data elements */

    switch (event_action(event) {
        case ACTION_PASTE:
            /* Initiate a blocking request, and get the data */
            data = (char *) xv_get(sel_requestor,
                                   SEL_DATA, &length, &format);
            /* returns with length set to SEL_ERROR          */
            /* if the if request fails.                      */
            break;
    }
}
```

Above, the attribute SEL_DATA with xv_get() sends the selection request to the selection owner. The returned data is placed in *data*. If the conversion was successful, the application needs to free the returned data, data when it is finished using it.

18.2.3.5 Requesting the CLIPBOARD selection—non-blocking

If a selection requestor wants to make a non-blocking request, it calls the procedure sel_post_req() which is defined by the selection package. This procedure sends a non-blocking request to the selection owner.

The procedure sel_post_req() has the following format:

```
int
sel_post_req( sel_req )
     Selection_requestor    sel_req;
```

The procedure sel_post_req() returns XV_OK or XV_ERROR. If no reply procedure is defined, XV_ERROR is returned. Example 18-2 shows how the selection requestor requests the clipboard selection using a non-blocking request.

Example 18-2. Non-blocking selection request

```
long                *file_buffer;     /* contents of selected file */
long                 file_buffer_size; /* # of longs in file_buffer */
Selection_requestor  sel_requestor;

main()
{
          .
          .
          .
     sel_requestor = xv_create(window, SELECTION_REQUESTOR,
                                SEL_REPLY_PROC, ReplyProc,
                                SEL_RANK_NAME, "CLIPBOARD",
                                NULL);

}
window_event_handler(..., event, ...)

Event          *event;

{

     switch (event_action(event) {
     case ACTION_PASTE:
              /* initiate non-blocking request */
              sel_post_req(sel_requestor);
              break;
          }
}
```

18.2.4 Converting the Selection (Selection Owner)

When the selection requestor posts a request, the request is sent via the X server to the client which currently owns the selection.* If the request is sent to an XView-based application, the toolkit will determine the appropriate selection-owner object to forward the request to. If the selection-owner object has a conversion procedure registered (see SEL_CONVERT_PROC), it will be called with information about the request. If the selection-owner object does not have a conversion procedure defined, but does have selection items registered on behalf of the selection owner, the toolkit will determine if the request matches any of the selection items. If a match is made, the toolkit will respond to the request with the selection item's data. If a match is not made, the toolkit will reject the request.

If a selection-owner object uses selection items for all targets for which it is willing to respond to, then there is no need to register a conversion procedure. If the selection-owner object uses selection items for some responses and converts other responses in a conversion procedure, then the default selection conversion procedure (`sel_convert_proc()`) must be called from within the selection-owner object's conversion procedure. This is where the selection-owner object's selection items are handled.

It is the responsibility of the selection owner to either convert the data into the type specified by the requestor or reject the request. The conversion of data typically happens in the selection-owner object's conversion procedure (see SEL_CONVERT_PROC). The conversion procedure should convert the selection it holds to the requested type. The conversion procedure's return value is set to TRUE if the owner successfully converts the selection to the target type and to FALSE if it rejects the selection request.

The form of the conversion procedure is:

```
int
convert_proc( sel, replyType, replyBuff, length, format )
       Selection_owner  sel;
       Atom            *replyType;
       Xv_opaque       *replyBuff;
       unsigned long   *length;
       int             *format;
```

The argument *sel* specifies the selection owner. When the conversion routine is called, *replyType* is set to the target requested. On return, it should indicate the type the selection was converted to. For example, the selection requestor may be requesting the target LENGTH so *replyType* would have the value LENGTH when the conversion routine is called. Before the conversion routine returns, *replyType* would be set to INTEGER, representing the type of the value returned. As another example, if the requestor is requesting the target FILE_NAME, the conversion procedure needs to explicitly set *replyType* to the atom which describes the converted type, in this case, TEXT.

The *replyBuff* is a pointer to a buffer address which contains the converted data. Do *not* return automatic storage in *replyBuff*. If your selection data uses more data than a long

*For performance reasons, local selection transfers using `sel_post_req()` do not go through the server, and the data is transfered within the application itself.

`int`, be sure to malloc the storage for the converted data. This storage can later be freed in a `SEL_DONE_PROC`.

The conversion procedure is called with *length* set to the server's maximum allowed buffer size. Before the conversion procedure returns, *length* should be assigned the number of elements in *replyBuff*.

format specifies a pointer to the element size of the data. *format* should be set to the element size used in `replyBuff`. Valid values are 8, 16, or 32 for 8-bit, 16-bit, or 32-bit, quantities, respectively.

All applications supporting selections are required by the ICCCM to convert the `TARGETS` type. This type returns a list of types the conversion procedure understands and supports. The conversion procedure should include the code necessary to convert this target (see ICCCM for more details).

For selections that are larger than the maximum size allowed by the server, the selection package sends the data in increments. The selection package handles this "large" selection case automatically. That is, if the conversion procedure returns a *replyBuff* larger than the maximum size allowed by the server, the selection package will divide the buffer into reasonable sizes and send the data in increments using the `INCR` target described by the ICCCM. If you want to specify that a selection should be transferred incrementally, you may specify an incremental transfer. An incremental transfer is specified by constructing an increment message. The increment message and incremental transfer are covered in Section 18.4, "How to Send Data Incrementally."

18.2.4.1 The default conversion procedure

The selection owner has a default conversion procedure called `sel_convert_proc()`. This conversion procedure is predefined to convert `TARGETS` for several targets whose conversion is supported by the selection package. This procedure also handles conversions where selection items are used (see Section 18.3.1, "The Selection Item").

Normally, an application-defined procedure should call `sel_convert_proc()` before returning. This assures that the selection-owner object will handle conversions involving a selection-item object. If you are sure that your conversion procedure handles all the targets that your application will support, then you do not need to call `sel_convert_proc()` from your conversion procedure.

18.2.4.2 Sample selection owner with conversion procedure

Example 18-3 shows a selection-owner object and defines the conversion procedure, `convert_proc()`. When an `ACTION_COPY` event is detected, `sel_owner` acquires the selection by setting the attribute `SEL_OWN` to `TRUE`. At this time you should also set `SEL_TIME` to the time on to the time of the `ACTION_COPY` event. If you do not set the selection time using `SEL_TIME`, the selection owner package sets the time, but this requires the package to perform additional processing.

The sample procedure `convert_proc()` is invoked when a selection requestor posts a request (not shown here). In this routine, the selection rank is stored in *selection* and the response data, "Primary Selection content . . . " is is stored in *str*. Lastly, *format*, *length*, *type*, and *replyBuff* are assigned the appropriate values to convert the data (*str*) to the requested type (`XA_INTEGER` or `XA_STRING`).

Example 18-3. Selection owner program

```
static int convert_proc();
Selection_owner     sel_owner;

main()
{
     :
     sel_owner = xv_create(window, SELECTION_OWNER,
                SEL_CONVERT_PROC, convert_proc,
                NULL);
     :
}

window_event_handler(..., event, ...)
      :
    Event          *event;
      :
{
      :
     switch (event_action(event) {
     case ACTION_COPY:
          xv_set(sel_owner, SEL_OWN, TRUE, NULL);
          break;
         }
      :
}

int
convert_proc( sel_owner, type, replyBuff, length, format )
    Selection_owner    sel_owner;
    Atom              *type;
    Xv_opaque         *replyBuff;
    unsigned long     *length;
    int               *format;
{
    int    len;
    Atom   selection;
    char   str[50];

    selection = (Atom) xv_get( sel_owner, SEL_RANK );
    strcpy( str,"Primary selection content...\n");

    if ( *type == TARGETS )  {
       /* Support target conversion here */
       return( TRUE );
    }

    if ((selection == XA_PRIMARY ) && ( *type == XA_STRING )) {
        *format = 8;
        *length = strlen( *str ) + 1 ;
```

Example 18-3. Selection owner program (continued)

```
        *type = type;
        *replyBuff = (Xv_opaque) strcpy(str);
        return(TRUE);
     }

    if ((selection == XA_PRIMARY ) && ( *type == XA_INTEGER )) {
        len = strlen(str);
        *format = 32;
        *length = 1;
        *replyBuff = (Xv_opaque)&len;
        return(TRUE);
    }
    /*   call sel_convert_proc(sel_owner, type,
                               replyBuff, length, format ); */
    /* return, or call the default conversion procedure */

}
```

18.2.5 Handling the Response (Selection Requestor)

Once the selection owner has responded to the request, the selection-requestor object is notified. If the selection-requestor object used a blocking request, then the blocked xv_get() will return with the response. If the requestor had made a non-blocking request then the response would be sent to the requestor's callback procedure. This callback procedure is called a reply procedure and can be registered on the requestor object using the SEL_REPLY_PROC attribute.

Information passed into the reply procedure will indicate whether the selection request was completed successfully. If it was successful, the requestor is free to use the response as appropriate. Typically, it will display the data in a window. The format for the application-defined reply procedure is:

```
void
reply_proc( sel_req, target, type, replyValue, length, format )
     Selection_requestor  sel_req;
     Atom                 target;
     Atom                 type;
     Xv_opaque            replyValue;
     unsigned long        length;
     int                  format;
```

The argument *sel_req* is the selection requestor. The name of the request/response is supplied in *target*. For example, `TARGETS`, `STRING`, or `LENGTH`. The argument *type* specifies the type of the target returned, values corresponding to the three mentioned, would be `TARGETS`, `STRING` and `INTEGER`. *replyValue* specifies the data content returned. *length* specifies the length of the data. *format* specifies the format of the data. Example 18-4 shows a sample reply procedure.

The data for *reply_Value* is malloced. The application is responsible for freeing this data once it is finished using it.

If the response was returned in increments, then the reply procedure will be called back more than once with individual buffers representing part of the response. The reply procedure must be able to handle incremental responses. See Section 18.4, “How to Send Data Incrementally,” for more information on how to do this.

In the case where multiple targets are requested, the selection’s reply procedure is called more than once.

Example 18-4. Sample reply procedure – SelectionReplyProc

```
void
SelectionReplyProc(sel, target, type, value, length, format)
    Selection_requestor   sel;
    Atom                target;
    Atom                type;
    Xv_opaque           value;
    unsigned long       length;
    int                 format;
{
    if (length == SEL_ERROR) {
        SelectionError(sel, target, *(int *)value);
        return;
    }

    if (target == ATOM(server, "TARGETS")) {
     textsw_insert(textsw,
               "Holder will convert the following targets:\n", 43);
                /* 43 is the size of the string */
      do {
        Atom *targets = (Atom *)value;
        char *target_name;

        if (targets[--length]) {
            target_name = (char *)xv_get(server, SERVER_ATOM_NAME,
                                targets[length]);

            textsw_insert(textsw, "\t", 1);
            textsw_insert(textsw, target_name, strlen(target_name));
            textsw_insert(textsw, "\n", 1);
        }
      } while(length);

    }
    else
    if (target == ATOM(server, "TIMESTAMP")) {
        char buf[10];

        textsw_insert(textsw, "TIMESTAMP of acquisition: ", 26);
        sprintf(buf, "%U\n", *(unsigned long *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "LENGTH")) {
        char buf[10];

        textsw_insert(textsw, "Length of selection: ", 21);
        sprintf(buf, "%d\n", *(int *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
```

Example 18-4. Sample reply procedure – SelectionReplyProc (continued)

```
    if (target == ATOM(server, "STRING")) {
        static int   incr = False;

        if (type == ATOM(server, "INCR")) {
            textsw_insert(textsw, "Contents of the selection:\n", 27);
            incr = True;
        }
        else
        if (length) {
            if (!incr)
                textsw_insert(textsw, "Contents of the selection:\n", 27);
            textsw_insert(textsw, (char *)value, length);
            textsw_insert(textsw, "\n", 1);
        }
         else
            incr = False;
    }
    else
    if (target == ATOM(server, "DELETE")) {
        textsw_insert(textsw, "The Selection has been deleted\n", 31);
    }

    textsw_insert(textsw, LINE, strlen(LINE));
}
```

18.2.5.1 Handling selection reply procedure errors

If the selection conversion fails, and you are using a non-blocking request, the reply procedure is called with *replyValue* set to an error code and *length* set to `SEL_ERROR`. The reply procedure error codes are defined in *sel_pkg.h* and shown in Table 18-2.

If the selection conversion fails, and you are using a blocking request without a reply procedure defined, the `xv_get()` returns with the *length* argument set to `SEL_ERROR`, *format* set to 0, and a `NULL` value is returned. If a reply procedure is defined, its *replyValue* argument is set to one of the error codes shown in Table 18-2.

Table 18-2. Error Codes

Code	Description
`SEL_BAD_CONVERSION`	If the conversion is refused by the selection holder or there is no holder of the selection, this value is returned. This may mean that there is no owner for the selection, that the owner does not support the conversion implied by target, or that the server did not have sufficient space.
`SEL_BAD_TIME`	The SelectionNotify time does not match the package time value.
`SEL_BAD_WIN_ID`	The SelectionNotify requestor ID does not match the package requestor ID.

Table 18-2. Error Codes (continued)

Code	Description
SEL_BAD_PROPERTY	Obsolete.
SEL_BAD_PROPERTY_EVENT	Obsolete.
SEL_PROPERTY_DELETED	Obsolete.
SEL_TIMEDOUT	Selection timed out.

A sample application-defined errror routine, for a non-blocking error handler, is shown in Example 18-5. Code similar to this should be invoked to handle errors in your selection application.

Example 18-5. Sample selection reply error handler

```
SelectionError(sel, target, errorCode)
    Selection_requestor    sel;
    Atom         target;
    int               errorCode;
{
    Atom     rank;
    char    *rank_string;
    char    *target_string = (char *)xv_get(server,
                                        SERVER_ATOM_NAME, target);
    char     msg[100];

    rank = (Atom)xv_get(sel, SEL_RANK);
    rank_string = (char *)xv_get(server, SERVER_ATOM_NAME, rank);

    sprintf(msg, "Selection failed for rank ‘‘%s’’ on target ‘‘%s’’: ",
                                     rank_string, target_string);
    textsw_insert(textsw, msg, strlen(msg));

    switch(errorCode) {
      case SEL_BAD_CONVERSION :
        textsw_insert(textsw, "Conversion Rejected",
        strlen("Conversion Rejected"));
        break;

      case SEL_BAD_TIME:
        textsw_insert(textsw, "Bad Time Match",
                  strlen("Bad Time Match"));
        break;

      case SEL_BAD_WIN_ID:
        textsw_insert(textsw, "Bad Window Match",
                  strlen("Bad Window Match"));
        break;

      case SEL_TIMEDOUT:
        textsw_insert(textsw, "Timeout", strlen("Timeout"));
        break;
    }
}
```

18.2.6 If the Selection is Lost (Selection Owner)

If the owner loses the selection, because someone else acquired the same rank selection, the lose procedure is called. The selection-owner object attribute `SEL_LOSE_PROC` specifies the selection owner's lose procedure. This procedure is called by the toolkit, to inform the selection owner that it has lost the ownership of the given selection. The form of the lose procedure is shown below:

```
void
lose_proc( sel )
    Selection_owner  sel;
```

sel specifies the selection-owner object.

One example of `lose_proc()` usage is for unhighlighting the highlighted text in a `textsw`. The code fragment below shows a selection lose procedure that simply informs the user that the selection was lost.

```
void
SelectionLoseProc(sel)
    Selection_owner   sel;
{
    xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
}
```

18.2.7 Cleanup – When the Selection Completes (Selection Owner)

The attribute `SEL_DONE_PROC` specifies the application-defined *done* procedure that is called by the toolkit when a selection request has successfully completed. It is called once following each successful transfer of data to the requestor. Thus, if the selection request is a multiple, or results in an incremental reply, the done procedure is called more than once. This procedure can be used to deallocate any selection replies allocated in the conversion procedure.

```
void
done_proc( sel, replyBuff, target )
     Selection_owner  sel;
     Xv_opaque        replyBuff;
     Atom             target;
```

The argument *sel* specifies the selection-owner object. *replyBuff* specifies the address which contains the converted data. *target* specifies the target type returned by the conversion procedure.

Example 18-6 shows a sample done procedure.

Example 18-6. Sample done procedure – SelectionDoneProc

```
void
SelectionDoneProc(sel, data, target)
    Selection_owner   sel;
    Xv_opaque         data;
    Atom              target;
{
    if (target == ATOM(server, "STRING"))
       free((char *)data);
}
```

18.3 How Selection Works (With a Selection Item)

This section describes using a selection item that can eliminate the selection owner's need for a conversion procedure. This may simplify the programmer's job of implementing a selection transfer.

Selection items are good to use for conversions that involve static data—data that will not change over the lifetime of the selection. Do not use a selection item for selections where the data associated with the selection is not known until the selection request is received (using a selection conversion procedure is best for this case). Also, do not use a selection item for any selection conversion that has side effects (for example, a delete request).

Below is a brief overview of the steps that take place during a single transfer of data from one XView application to another using a selection item. We'll assume we have two applications, application *A* and application *B*, either of which can operate as the owner or the requestor.

1. (Owner side.) The user makes a selection in application *A* and the application highlights the selection.
2. (Owner side.) The user may pre-register a conversion for a particular target by creating a selection-item object and setting its attributes.
3. (Owner side.) In the XView Selection package, nothing further happens until an event of interest is detected, for example, an `ACTION_COPY` event. At this time, a selection item may claim ownership of the current selection in application *A*. Also at this time, selection data may be attached to the selection-item object.
4. (Requestor side.) The user pastes the selected data into application *B*, causing an `ACTION_PASTE` event to occur.
5. (Requestor side.) The `ACTION_PASTE` event may cause the Selection Requestor to make a non-blocking request to receive the selected data into application *B*.

6. (Owner side.) When there is a match between a selection-item object's target type, and a requested target, the selection package converts and sends the selection data to the selection requestor. XView internally handles the data conversion by calling the package's default conversion procedure `sel_convert_proc()`.
7. (Requestor side.) Once the data conversion has completed (either successfully or unsuccessfully), the application-defined reply procedure is invoked. The data may displayed in application *B* (if the conversion routine was successful) or the user may simply be given some indication that data was received. Other possibilities include: the kind of data selected in application *A* cannot be pasted in application *B*, or that the kind of data requested by *B* cannot be supplied by *A*.
8. (Owner side.) Once the selection transfer is complete, an application-defined "done" procedure may be called by the selection owner. The done procedure may be used to free the memory associated with the selection, or to perform other cleanup that is required.
9. (Owner side.) If another application acquires the selection, application *A*'s "lose" procedure is called. This procedure is used to handle notification of loosing selection ownership. It tells the selection owner (application *A*) that it has lost ownership of the selection. For example, the lose procedure might unhighlight text that was previously selected and highlighted.

18.3.1 The Selection Item

The first task for the application that is to become the selection owner, using a selection-item object, is to pre-register a conversion or several conversions. Below, we describe this step in detail. Also, the selection needs to be marked. For example, when the pointer is placed over text in a canvas, an `ACTION_SELECT` should highlight the selection. While `LOC_DRAG` occurs, the highlight would be extended over the selection.

After the user makes a selection and it is marked, the application waits to receive an event of interest. For example, in a text subwindow, the act of selecting an item causes the primary rank to be acquired. For the text subwindow example, an `ACTION_COPY` event causes the CLIPBOARD selection to be acquired (note that in this case two selections are acquired since the COPY operation in OPEN LOOK uses the clipboard).

With a selection item, applications perform three steps to become the selection owner:

1. A `Selection_owner` object must be created using `xv_create()` (assuming a previously created selection-owner object is *not* being re-used.) A selection-item object is created and its attributes are set.
2. The selection must be acquired.
3. The selection data is associated with the selection item.

A selection-owner object is created using `xv_create()`:

```
Selection_owner   sel_owner;
sel_owner = xv_create(window, SELECTION_OWNER,
                      NULL);
```

To pre-register a selection item for a string conversion, use the following code:

```
Selection_item   sel_item;
sel_item = xv_create(sel_owner,
                 SELECTION_ITEM,
                 SEL_TYPE_NAME,    "STRING",
                 SEL_FORMAT,       8,  /* bits per unit (char)*/
                 NULL);
```

Pre-register a conversion by setting several selection-item object attributes. `SEL_TYPE_NAME` specifies the type, using a string for the atom type of the selection. `SEL_FORMAT` specifies the format for the data. Valid values are 8, 16, or 32 for 8-bit, 16-bit, or 32-bit quantities, respectively. `SEL_TYPE` specifies the type of the conversion that the item supports. To support more than one type, you need to use additional selection items, one for each type. Since a string conversion is the default for a selection item, the following code registers the same conversion at that shown above:

```
Selection_item     sel_item;
sel_item = xv_create(sel_owner, SELECTION_ITEM,
                     NULL);
```

The owner of a selection-item object is an object of type `Selection_owner`. The owner of a selection item defines the rank to which the item belongs. Figure 18-4 shows the class hierarchy for a selection-item object.

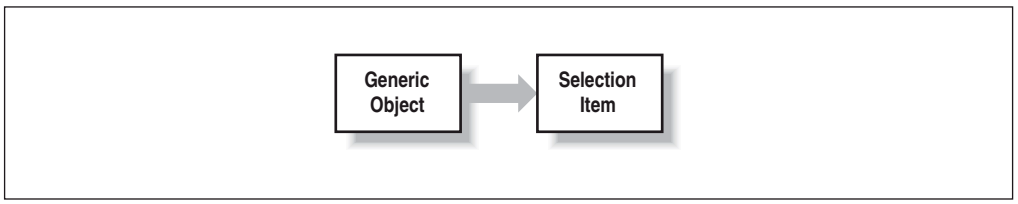

Figure 18-4. Selection item class hierarchy

A selection is acquired by setting `SEL_OWN` to `TRUE` on the selection-owner. By default, the selection is associated with the primary rank (`XA_PRIMARY`). If your selection needs to use another rank, use either `SEL_RANK` or `SEL_RANK_NAME` to specify the rank as shown below:

```
xv_set(sel_owner, SEL_OWN, TRUE,
                 SEL_RANK_NAME "CLIPBOARD",
                 NULL);
```

After the selection-owner object and the selection-item object are created, and the selection is acquired, data needs to be associated with the selection item (if the selection item is defined, its data can be set at any time). If the selection is a string transfer using the primary selection, this step is easy:

```
char  *sel_string;

sel_string="Sample selection data.";

xv_set(sel_item,
    SEL_DATA, sel_string, /*pointer to highlighted data*/
    SEL_LENGTH, strlen(sel_string),
    NULL);
```

The attribute `SEL_DATA` associates the selection data with the selection-item object. `SEL_LENGTH` specifies the number of data elements in the selection item. Another selection item attribute, `SEL_COPY` indicates whether the selection item package should copy and maintain the selection-item object's data. If this is set to `FALSE`, it is up to the application to maintain the data. To pre-register a selection item for the targets type that returns a list of five atoms, use the following code:

```
Selection_item  sel_item;
Atom            targets[5];
/* initialize the targets array */
targets[0] = (Atom)xv_get(server, SERVER_ATOM, "TARGETS");
targets[1] = (Atom)xv_get(server, SERVER_ATOM, "TIMESTAMP");
targets[2] = (Atom)xv_get(server, SERVER_ATOM, "LENGTH");
targets[3] = (Atom)xv_get(server, SERVER_ATOM, "STRING");
targets[4] = (Atom)xv_get(server, SERVER_ATOM, "DELETE");

sel_item = xv_create(sel, SELECTION_ITEM,
                         SEL_TYPE_NAME,    "TARGETS",
                         SEL_FORMAT,       32,
                         SEL_LENGTH,       5,
                         SEL_DATA,        targets,
                         NULL);
```

Once a selection-item object is created, and the selection data is associated with the selection item, the selection item's owner waits to receive a request. When and if the item's owner receives a selection request, the selection data is converted, according to the specifications set using the selection item attributes. Data is supplied through the `SEL_DATA` attribute. Note that the default conversion procedure `sel_convert_proc()` needs to be called in order to convert a selection using a selection item.

The owner of a selection-item object, is a selection-owner object. A selection-owner object can access its selection items using the attributes `SEL_FIRST_ITEM` and `SEL_NEXT_ITEM`.

18.4 How to Send Data Incrementally (Selection Owner)

Selections whose data is larger than the server's maximum request size require special handling. The selection package checks these limits and handles these large cases internally, using incremental selections. The selection owner's conversion procedure may also specify, for any reason, that the data is to be sent in increments.

An incremental transfer is specified by constructing an increment message as follows:

1. Set *replyType* to an atom named `INCR`.
2. Set *replyBuff* to an integer representing a lower bound on the number of bytes of data in the selection.
3. Set *length* to 1.
4. Set *format* to 32.

The conversion routine should then return `TRUE`. The selection package sends the `INCR` message to the requestor and then calls the conversion routine repeatedly, for each buffer making up the response, until *length* is set to zero by the application. This indicates the end of the data transfer.

A selection owner can determine if a selection is larger than the maximum allowed by the server by comparing the value of the length variable. The conversion routine is called with *length* set to the server's maximum request size. If the owner has chosen to send the selection data in increments, the size of each increment should be less than or equal to *length*. If the size is larger, the selection package sends the data as a series of incremental transfers. Example 18-7 shows a conversion procedure that sends data incrementally.

Example 18-7. An incremental conversion procedure

```
static int
ConvertProc( selOwner, type, data, length, format )
Selection_owner    sel_owner;
    Atom          *type;
    Xv_opaque     *data;
    long          *length;
    int           *format;
{
    static char  *tmp=(char *) NULL;
    static int  firstTime=1;
    static int  numBytes=0;

    fileSize = TERM_CAP_SIZE;

    if (  *type == XA_STRING )   {
          /*
           * Send INCR message to the requestor.
          */
          if ( firstTime ) {
              static long   fSize;

              fSize = TERM_CAP_SIZE;
              *type = xv_get( server, SERVER_ATOM, "INCR");
```

Example 18-7. An incremental conversion procedure (continued)

```
            *data = (Xv_opaque) &fSize;
            *format = 32;
            *length = 1;
            firstTime = 0;
            return TRUE;
        }

        if ( tmp != NULL )
             free( tmp );

        tmp = (char *) malloc( BUFSIZE  );

        if( (numBytes = read( fd, tmp, BUFFSIZE )) == -1 ) {
            fprintf( stderr,"errno = %d \n", errno );
            numBytes=0;
        }
        if( (numBytes == 0 ) {
            free(tmp)
            tmp=null
        }
        *format = 8;
        *length = numBytes;
        *type = XA_STRING;
        *data = (Xv_opaque) tmp;
        return TRUE;
    }
    return FALSE;
}
```

18.4.1 How to Handle Incremental Replies (Selection Requestor)

The selection package starts an incremental reply by sending the `INCR` type to the reply procedure. *replyValue* is set to a lower bound on the number of bytes of data in the selection. The reply procedure needs to be able to build up the data as it is called repeatedly until all the selection data has been transferred.

The conversion procedure informs the reply procedure when the incremental transfer is complete by sending a message. The conversion procedure sets *length* to zero and *replyValue* to `NULL`, indicating the end of incremental data transfer. Clients need to free the *reply Value*. Example 18-8 shows how the reply procedure handles incremental replies.

Example 18-8. Incremental reply – IncrReply.c

```
static void
ReplyProc( selReq, target, type, replyBuf, len, format )
     Selection_requestor  selReq;
     Atom                 target;
     Atom                 type;
     Xv_opaque            replyBuf;
     unsigned long        len;
     int                  format;
{
```

Example 18-8. Incremental reply – IncrReply.c (continued)

```
    if ( len == SEL_ERROR ) {
        int   errCode;

        bcopy( (char *) replyBuf, (char *) &errCode, sizeof( int ) );
        switch( errCode ) {
            case SEL_BAD_CONVERSION :
                printf("ReplyProc: Conversion failed!\n");
                break;
            case SEL_BAD_TIME:
                printf("ReplyProc: Bad time!\n");
                break;
            case SEL_BAD_WIN_ID:
                printf("ReplyProc: Bad window id!\n");
                break;
            case SEL_TIMEDOUT:
                printf("ReplyProc: Timed out!\n");
                break;
        }
    }

    if ( len == 0 ) {
       printf("End of incremental data transfer.\n");
       return;
    }

    if ( type == xv_get( server, SERVER_ATOM, "INCR") ) {
       long  size;

       bcopy( (char *) replyBuf, (char *) &size, sizeof( long ) );
       printf("Get ready for INCR of size %d\n", size );
    }

    if ( ( type == XA_STRING ) && len )
       printf("%.*s\n", len, (char *) replyBuf );
}
```

18.5 Requesting and Converting Multiple Targets

A selection request may involve more than one target. This type of request is called a `MULTIPLE` request. Normally on both the selection owner side, and on the selection requestor side, a `MULTIPLE` request is handled by the packages, without any interaction by the application programmer. This section outlines how the selection-requestor package and the selection-owner package handle `MULTIPLE` requests.

The selection requestor package detects that more than one target has been requested when the following attributes are used to request more than a single target:

```
SEL_APPEND_TYPE
SEL_APPEND_TYPES
SEL_TYPES
SEL_TYPE_NAMES
```

The selection requestor then requests that the selection owner convert a list of targets.

On the selection owner side, a `MULTIPLE` request is treated as a stream of single requests. The package calls the conversion procedure as many times as required to convert all of the targets requested.

In the special case where a multiple request also involves an incremental target (`INCR`), the selection owner calls the application-defined conversion procedure with *format* set `SEL_MULTIPLE` to indicate that the selection request is a incremental request that is part of a multiple request.

18.6 Additional Transfer Mechanisms (Selection Requestor)

A selection requestor may associate data with the window property being used in the selection transaction. This property may be used by the selection owner for certain types of selection transfers. The selection-requestor object attributes: `SEL_PROP_DATA`, and `SEL_TYPE_INDEX` support bi-directional data transfer for a property associated with a selection. In addition, the attributes `SEL_PROP_FORMAT`, `SEL_PROP_LENGTH`, `SEL_PROP_TYPE`, and `SEL_PROP_TYPE_NAME`, also support this type of selection transfer. A selection requestor may associate a property with the selection that the requestor is requesting. For example, this associated property may be used to support an `INSERT_SELECTION` or a `INSERT_PROPERTY` target. Or properties may be used by the owner for other reasons. A selection-requestor object may associate data with the property that the selection is requesting. This information may be of interest to a selection owner for certain types of selection transfers.

18.7 Additional Transfer Mechanisms (Selection Owner)

If the selection requestor is sending data to the selection owner, using additional properties, the attribute `SEL_PROP_INFO` allows the selection owner to access this property data.

18.8 Sample Selection Owner Program with a Selection Item

Example 18-9 shows a simple panel with two text items and two buttons inside a frame. This example uses a selection item to return the TARGETS type and a conversion procedure to return the remaining supported types. The “Selection:” field allows the user to enter the desired selection rank. The “Contents:” field allows the user to enter a string, the selection data to be operated on. The “Own Selection” button causes the application to own the selection. The “Lose Selection” button causes the selection to be lost.

Example 18-9. Sample program – sel_hold.c

```
/*
 * sel_hold.c: Example of how to acquire and hold a selection.
 *
 */
#include <stdio.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/font.h>
#include <xview/sel_pkg.h>

Frame             frame;
Xv_Server         server;
Panel             panel;
Panel_item        p_selection,
                  p_contents,
                  p_own,
                  p_lose;

Selection_owner   sel;
Selection_item    sel_targets;

#define ATOM(server, name)  (Atom)xv_get(server, SERVER_ATOM, name)

main(argc, argv)
    int    argc;
    char **argv;
{
    Panel_setting   NotifyProc();
    int             SelectionConvertProc();
    void            SelectionDoneProc(),
                    SelectionLoseProc();
    Xv_Font         font;
    Atom            targets[5];

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create((Window)NULL, FRAME,
              XV_X,      520,
              XV_Y,      655,
              XV_LABEL,  "Selection Holder Example",
              FRAME_SHOW_FOOTER,  True,
              NULL);

    panel =  xv_create(frame, PANEL, NULL);


    p_selection = xv_create(panel, PANEL_TEXT,
                PANEL_LABEL_STRING,        "Selection:",
                PANEL_VALUE_DISPLAY_LENGTH, 40,
                PANEL_NOTIFY_PROC,          NotifyProc,
                PANEL_ITEM_X,               xv_col(panel,0),
                PANEL_ITEM_Y,               xv_row(panel,0),
                NULL);
```

Example 18-9. Sample program – sel_hold.c (continued)

```
    p_contents = xv_create(panel, PANEL_TEXT,
                  PANEL_LABEL_STRING,        "Contents:",
                  PANEL_VALUE_DISPLAY_LENGTH, 40,
                  PANEL_ITEM_X,               xv_col(panel,0),
                  PANEL_ITEM_Y,               xv_row(panel,1),
                  NULL);

    p_own = xv_create(panel, PANEL_BUTTON,
                  PANEL_LABEL_STRING, "Own Selection",
                  PANEL_NOTIFY_PROC,   NotifyProc,
                  PANEL_ITEM_X,        xv_col(panel,5),
                  PANEL_ITEM_Y,        xv_row(panel,2),
                  NULL);

    p_lose = xv_create(panel, PANEL_BUTTON,
                  PANEL_LABEL_STRING, "Lose Selection",
                  PANEL_NOTIFY_PROC,   NotifyProc,
                  PANEL_ITEM_X,        xv_col(panel,30),
                  PANEL_ITEM_Y,        xv_row(panel,2),
                  NULL);

     /* Create a selection owner object */
    sel = xv_create(panel, SELECTION_OWNER,
                  SEL_CONVERT_PROC,    SelectionConvertProc,
                  SEL_DONE_PROC,       SelectionDoneProc,
                  SEL_LOSE_PROC,       SelectionLoseProc,
                  NULL);

    targets[0] = (Atom)xv_get(server, SERVER_ATOM, "TARGETS");
    targets[1] = (Atom)xv_get(server, SERVER_ATOM, "TIMESTAMP");
    targets[2] = (Atom)xv_get(server, SERVER_ATOM, "LENGTH");
    targets[3] = (Atom)xv_get(server, SERVER_ATOM, "STRING");
    targets[4] = (Atom)xv_get(server, SERVER_ATOM, "DELETE");

     /* Create a selection item, owned by the selection owner we just
      * created.  This pre-registers a conversion, in this case a
      * conversion for ''TARGETS''.
      */
    sel_targets = xv_create(sel, SELECTION_ITEM,
                   SEL_TYPE_NAME,    "TARGETS",
                   SEL_FORMAT,       32,
                   SEL_LENGTH,       5,
                   SEL_DATA,        (Xv_opaque)targets,
                   NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
    exit(0);
}
```

The selection-owner object `sel` specifies the conversion procedure, the done procedure, and the lose procedure. Next, the targets array is filled with supported atoms. These are the names for the supported conversions. The selection-item object's conversion is specified

using the attributes: SEL_TYPE_NAME, SEL_FORMAT, and SEL_LENGTH. The attribute SEL_DATA associates the selection data with the selection item.

18.8.0.1 The notify procedure

Example 18-10 shows how the NotifyProc() routine is used to handle the selection rank entry, owning the selection, and losing the selection. If a selection rank has been entered, it is read from the panel text item and then assigned to sel, the selection-owner object. If the "own" button is pressed, the current selection is acquired by setting SEL_OWN to TRUE and SEL_TIME to the time of the event. If the "lose" button is pressed, the current selection is released by setting SEL_OWN to FALSE and SEL_TIME to the time of the event.

Example 18-10. The notify procedure – NotifyProc()

```
Panel_setting
NotifyProc(item, event)
    Panel_item       item;
    Event           *event;
{
    if (item == p_selection) {
       char     *rank;

         /* Get the rank of the selection     */
         /* the user would like to use.       */
         rank = (char *)xv_get(item, PANEL_VALUE);

         /* Set the rank to our selection owner object. */
         xv_set(sel, SEL_RANK_NAME, rank, NULL);

        return(PANEL_NEXT);

    } else if (item == p_own) {

         /* The user pressed the ''own'' button, so we */
         /* acquire the selection.                     */
         xv_set(sel, SEL_OWN, True,
                     SEL_TIME, event_time(event),
                     NULL);

         xv_set(frame,
                FRAME_LEFT_FOOTER, "Acquired Selection...", NULL);

    } else if (item == p_lose) {

         /* The user pressed the ''lose'' button, so we lose ownership
          * of the selection.
          */
         xv_set(sel, SEL_OWN, False,
                     SEL_TIME, event_time(event),
                     NULL);

         xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
    }

    return(PANEL_DONE);
}
```

18.8.0.2 The conversion procedure

The conversion procedure is specified by the SEL_CONVERT_PROC attribute. This conversion procedure, SelectionConvertProc, is called whenever an application makes a request to the selection owned by this application. This procedure is shown in Example 18-11. The selection owner responds to the requesting application by converting the acquired selection to the target type requested. A well behaved routine provides for the case of a TARGETS request by returning a list of valid target types to the requestor. In this case, the selection owner returns the length of the string selected for LENGTH, a copy of the contents of the string selected for STRING, and sets the string to NULL and returns NULL for the DELETE target.

Example 18-11. Sample conversion procedure – SelectionConvertProc

```
int
SelectionConvertProc(sel, target, data, length, format)
    Selection_owner    sel;
    Atom              *target;  /* Input/Output */
    Xv_opaque         *data;    /* Output */
    unsigned long     *length;  /* Output */
    int               *format;  /* Output */
{

    /* Request for the length of the selection. */
    if (*target == ATOM(server, "LENGTH")) {
          static unsigned long  len;
          char                  *contents;

          contents = (char *)xv_get(p_contents, PANEL_VALUE);
          len = strlen(contents);

          *target = ATOM(server, "INTEGER");
          *format = 32;
          *length = 1;
          *data = (Xv_opaque)&len;
          return(True);
    }
    /* Request for the string contents of the selection. */
    else if (*target == ATOM(server, "STRING")) {
          char                   *contents;

          contents = (char *)xv_get(p_contents, PANEL_VALUE);

          *target = ATOM(server, "STRING");
          *format = 8;
          *length = strlen(contents);
          *data = (Xv_opaque)strdup(contents);
          return(True);
    }
    /* Request to delete the selection. */
    else if (*target == ATOM(server, "DELETE")) {
          xv_set(p_contents, PANEL_VALUE, "", NULL);
          *target = ATOM(server, "NULL");
          *format = 32;
          *length = 0;
          *data = (Xv_opaque)NULL;
          return(True);
```

Example 18-11. Sample conversion procedure – SelectionConvertProc (continued)

```
    } else
    /* Call the default selection conversion procedure.
     * It handles requests for any pre-registered
     * conversions, including TARGETS.
     */
      return(sel_convert_proc(sel, target, data, length, format));
}
```

18.8.1 The Done Procedure

The selection done procedure is called after each conversion has happened. This gives the application a chance to free up memory.

```
void
SelectionDoneProc(sel, data, target)
    Selection_owner  sel;
    Xv_opaque        data;
    Atom             target;
{
    if (target == ATOM(server, "STRING"))
      free((char *)data);
}
```

Note, in this example, we only alloc data for string requests so we only free it for string data.

18.8.2 The Lose Procedure

The lose procedure lets the selection package lose ownership a selection gracefully. For example, the user can can be informed that the selection was lost, as in the following example.

```
void
SelectionLoseProc(sel)
    Selection_owner      sel;
{
    xv_set(frame, FRAME_LEFT_FOOTER, "Lost Selection...", NULL);
}
```

18.9 Sample Selection Requestor Program

Example 18-12 shows three panel items that are created to handle the selection rank, the target(s) request, and initiating the request. Below them is a `textsw` that displays the response to the request.

Example 18-12. Sample selection requestor program – sel_req.c

```
/*
 * sel_req.c: Example of how to make requests to a selection owner for
 *            the selection contents.
 */
#include <stdio.h>
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/font.h>
#include <xview/sel_pkg.h>

Frame          frame;
Textsw         textsw;
Xv_Server      server;
Panel          panel;
Panel_item     p_selection,
               p_target,
               p_request;

Selection_requestor  sel;

#define TARGETS         1<<0
#define TIMESTAMP       1<<1
#define LENGTH          1<<2
#define STRING          1<<3
#define DELETE          1<<4
#define LINE       "-----------------------------------------"

#define ATOM(server, name)   (Atom)xv_get(server, SERVER_ATOM, name)

main(argc, argv)
    int     argc;
    char  **argv;
{
    void       MakeRequest(),
               SelectionReplyProc(),
               RequestChoice();
    Xv_Font    font;

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = xv_create((Window)NULL, FRAME,
              XV_X,            10,
              XV_Y,            10,
              XV_LABEL,        "Selection Requestor Example",
              NULL);
```

Example 18-12. Sample selection requestor program – sel_req.c (continued)

```
    panel = xv_create(frame, PANEL,
                PANEL_LAYOUT,       PANEL_VERTICAL,
                NULL);

    p_selection = xv_create(panel, PANEL_TEXT,
                PANEL_LABEL_STRING,            "Selection:",
                PANEL_VALUE_DISPLAY_LENGTH,     40,
                PANEL_NOTIFY_PROC,              MakeRequest,
                NULL);

    p_target = xv_create(panel, PANEL_TOGGLE,
                PANEL_LABEL_STRING,      "Request:",
                PANEL_NOTIFY_PROC,        RequestChoice,
                PANEL_CHOICE_STRINGS,    "TARGETS",
                                         "TIMESTAMP",
                                         "LENGTH",
                                         "STRING",
                                         "DELETE",
                                          NULL,
                NULL);
    p_request = xv_create(panel, PANEL_BUTTON,
                PANEL_LABEL_STRING,     "Make Request",
                PANEL_NOTIFY_PROC,       MakeRequest,
                PANEL_ITEM_X,            xv_cols(panel, 20),
                NULL);

    window_fit(panel);

    textsw = xv_create(frame, TEXTSW,
                XV_X,          0,
                XV_FONT,       font,
                WIN_BELOW,     panel,
                NULL);

       /* Create a selection requestor object. */
    sel = xv_create(panel, SELECTION_REQUESTOR,
                SEL_REPLY_PROC,       SelectionReplyProc,
                NULL);

    window_fit(frame);
    xv_main_loop(frame);
    exit(0);
}
```

In the requestor application, a `PANEL_TEXT` item is created where the selection rank can be entered by the user. Next, a set of nonexclusive toggle buttons are created to allow the user to choose the target type for the conversion by the selection owner. A “Make Request” button is created that will send a request to the selection owner every time it is pressed. A `TEXTSW` is created to display the output of the request and lastly, a selection-requestor object, `sel`, is created.

Example 18-13 shows the notify procedure associated for the selection requestor example *sel_req.c*. It sets and/or appends selection types to the selection requestor based on the choices selected by the user.

Example 18-13. Sample requestor notify procedure

```
void
RequestChoice(item, value, event)
    Panel_item       item;
    unsigned int     value;
    Event           *event;
{
    int   set = False;

    /* Build the request based on the toggle items the user has selected. */

    if (value & TARGETS) {
        if (set)
              xv_set(sel, SEL_APPEND_TYPE_NAMES, "TARGETS", NULL, NULL);
           else
              xv_set(sel, SEL_TYPE_NAME, "TARGETS", NULL);
              set = True;
    }

    if (value & TIMESTAMP) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "TIMESTAMP", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "TIMESTAMP", NULL);
            set = True;
    }

    if (value & LENGTH) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "LENGTH", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "LENGTH", NULL);
        set = True;
    }

    if (value & STRING) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "STRING", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "STRING", NULL);
        set = True;
    }

    if (value & DELETE) {
        if (set)
            xv_set(sel, SEL_APPEND_TYPE_NAMES, "DELETE", NULL, NULL);
        else
            xv_set(sel, SEL_TYPE_NAME, "DELETE", NULL);
        set = True;
    }
}
```

The notify procedure, shown in Example 18-14, is associated with the “Make Request” button. It initiates the request by doing three things:

- Getting the selection rank.
- Setting the selection rank.
- Posting a selection request.

Example 18-14. Sample make request notify procedure

```
void
MakeRequest(item, event)
    Panel_item  item;
    Event       *event;
{
    if (item == p_selection) {
        char *rank = NULL;

        /* Set the rank of the selection we are */
        /* going to make requests to.           */
        rank = (char *)xv_get(item, PANEL_VALUE);
        xv_set(sel, SEL_RANK_NAME,        rank,
                        NULL);
    }
    else {
        /* Post a non-blocking request to the selection owner. */
        sel_post_req(sel);
        textsw_erase(textsw, 0, TEXTSW_INFINITY);
    }
}
```

18.9.0.1 Sample reply procedure

Example 18-15 shows the reply procedure for *sel_req.c*.

Example 18-15. Selection reply procedure

```
void
SelectionReplyProc(sel, target, type, value, length, format)
    Selection_requestor   sel;
    Atom                target;
    Atom                type;
    Xv_opaque           value;
    unsigned long       length;
    int                 format;
{
    if (length == SEL_ERROR) {
        SelectionError(sel, target, *(int *)value);
        return;
    }

    if (target == ATOM(server, "TARGETS")) {
      textsw_insert(textsw,
                "Holder will convert the following targets:\n", 43);
                 /* 43 is the size of the string */
```

Example 18-15. Selection reply procedure (continued)

```
      do {
        Atom *targets = (Atom *)value;
        char *target_name;

        if (targets[--length]) {
            target_name = (char *)xv_get(server, SERVER_ATOM_NAME,
                                 targets[length]);

            textsw_insert(textsw, "\t", 1);
            textsw_insert(textsw, target_name, strlen(target_name));
            textsw_insert(textsw, "\n", 1);
        }
      } while(length);

    }
    else
    if (target == ATOM(server, "TIMESTAMP")) {
        char buf[10];

        textsw_insert(textsw, "TIMESTAMP of acquisition: ", 26);
        sprintf(buf, "%U\n", *(unsigned long *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "LENGTH")) {
        char buf[10];

        textsw_insert(textsw, "Length of selection: ", 21);
        sprintf(buf, "%d\n", *(int *)value);
        textsw_insert(textsw, buf, strlen(buf));
    }
    else
    if (target == ATOM(server, "STRING")) {
        static int   incr = False;

        if (type == ATOM(server, "INCR")) {
            textsw_insert(textsw, "Contents of the selection:\n", 27);
            incr = True;
        }
        else
        if (length) {
            if (!incr)
                textsw_insert(textsw, "Contents of the selection:\n", 27);
            textsw_insert(textsw, (char *)value, length);
            textsw_insert(textsw, "\n", 1);
        }
         else
            incr = False;
    }
    else
    if (target == ATOM(server, "DELETE")) {
        textsw_insert(textsw, "The Selection has been deleted\n", 31);
    }

    textsw_insert(textsw, LINE, strlen(LINE));
}
```

18.9.0.2 Sample error procedure

The error procedure for this example is shown in Example 18-16.

Example 18-16. Sample error procedure – SelectionError.c

```
SelectionError(sel, target, errorCode)
    Selection_requestor    sel;
    Atom         target;
    int               errorCode;
{
    Atom     rank;
    char    *rank_string;
    char    *target_string = (char *)xv_get(server,
                                    SERVER_ATOM_NAME, target);
    char     msg[100];

    rank = (Atom)xv_get(sel, SEL_RANK);
    rank_string = (char *)xv_get(server, SERVER_ATOM_NAME, rank);

    sprintf(msg, "Selection failed for rank ‘‘%s’’ on target ‘‘%s’’: ",
                                    rank_string, target_string);
    textsw_insert(textsw, msg, strlen(msg));

    switch(errorCode) {
      case SEL_BAD_CONVERSION :
        textsw_insert(textsw, "Conversion Rejected",
        strlen("Conversion Rejected"));
        break;

      case SEL_BAD_TIME:
        textsw_insert(textsw, "Bad Time Match",
                  strlen("Bad Time Match"));
        break;

      case SEL_BAD_WIN_ID:
        textsw_insert(textsw, "Bad Window Match",
                  strlen("Bad Window Match"));
        break;

      case SEL_TIMEDOUT:
        textsw_insert(textsw, "Timeout", strlen("Timeout"));
        break;
    }
}
```

18.10 Selection Package Summary

Table 18-3 lists the selection procedures. Table 18-4 lists the selection attributes, including all the attributes for the `SELECTION` package, as well as the `SELECTION_OWNER`, `SELECTION_REQUESTOR`, and the `SELECTION_ITEM` attributes. These attributes and procedures are described fully in the *XView Reference Manual*.

Table 18-3. Selection Procedures

Procedure	Package
`sel_convert_proc()`	`Selection Owner`
`sel_post_req()`	`Selection Requestor`

Table 18-4. Selection Attributes

`SEL_APPEND_TYPE_NAMES`	`SEL_PROP_FORMAT`
`SEL_APPEND_TYPES`	`SEL_PROP_INFO`
`SEL_COPY`	`SEL_PROP_LENGTH`
`SEL_CONVERT_PROC`	`SEL_RANK`
`SEL_DATA`	`SEL_RANK_NAME`
`SEL_DONE_PROC`	`SEL_REPLY_PROC`
`SEL_FIRST_ITEM`	`SEL_TIME`
`SEL_FORMAT`	`SEL_TIMEOUT_VALUE`
`SEL_LENGTH`	`SEL_TYPE`
`SEL_LOSE_PROC`	`SEL_TYPE_INDEX`
`SEL_NEXT_ITEM`	`SEL_TYPE_NAME`
`SEL_OWN`	`SEL_TYPE_NAMES`
`SEL_PROP_DATA`	`SEL_TYPES`
`XV_XID`	