The X Selection Mechanism

or

How to Cut and Paste in 1000 lines or more

Keith Packard

X Consortium
Laboratory for Computer Science
Massachusetts Institute of Technology

ABSTRACT

While the existance of the selection mechanism in X may be wide known, the details of using it are not. This paper, while not a research paper per se, attempts to join the technical details with some practical experience. Along with this paper, a sample application was written which provides a working example of the ideas presented here.

1. What are Selections?

Selections are global server resources named by an atom and owned by a particular client. The number of selections is not limited by the protocol; as many selections as atoms may exist. Selections are designed to provide the basis for building communication mechanisms between clients. The official definition is found in the glosary of the X Protocol:

> "...an indirect property with dynamic type; that is, rather than having the property stored in the server, it is maintained by some client (the ‘‘owner’’). A selection is global in nature and is thought of as belonging to the user (although maintained by clients), rather than as being private to a particular window subhierarchy or a particular set of clients."

From the applications perspective, selections provide a mechanism for transmitting information between X clients. As X is a networking protocol, the existance of a separate channel for data transmission between the various clients cannot be assumed to exist. Selections are intended only for data transfer which directly relates the the user-interface aspects of the application, although there isn't any enforcement of this policy.

The data represented by selections, are not limited to strings of ASCII text. Selections remove all data interpretation from the X protocol and allow the applications to negotiate on the format in which the data will be represented ("with a dynamic type"). For example, a text editor may be able to provide substantial information about the format of the text included in the selection for use by other instances of the same editor, while different clients will still be able to negotiate for the plain text, stripped of the unwanted formatting data.

Selections provide this negotiable data format not by having the provider create all possible represetations of the data and store them all, but by having the selection name the owner of that selection ("an indirect property"). This information acts as a pointer to something which can produce whatever format is desired, sidestepping the whole issue of interchange formats by allowing as many different ones as you like, while not costing anything in storage space.

Another important problem which this addresses is that the server may not be able to provide storage for an arbitrary amount of client data. The ICCCM provides a system for transmitting large selections a small bit at a time, reducing the storage requirements in the server. Unfortunately, this frequently encourages the widespread abuse of the selection mechanism for bulk data transfer. It is more than possible to use the X protocol to implement a remote file system, or print spooler.

A major drawback of not storing the contents in the server, however, is that the contents of the selection disappear when the client which holds them is disconnected from the server. This means that selections should not be used for long-term storage; as that would force the selection provider to remain connected to the X server for that entire period. The ICCCM also responds to this limitation by providing the CLIPBOARD selection.

2. How selections help in providing seamless integration

Seamless integration is a pretty common marketing term, but the essential notion is sound: hide the divisions between disparate portions of the system so that the combination acts together in a cohesive manner. This helps not only the novice user by reducing the training required when learning new tasks, but also helps the experienced user by limiting the amoung of mental "gear shifting" requried when moving from task to task.

Reasonable use of selections can significantly enhance the seamlessness of the system. Because selections are relatively cheap to instantiate, as this doesn't involve bulk data transfer, applications can use the selection mechanism for data transfer even when the most frequent target of the transfer lies in the same address space as the source. By uniformly using the selection mechanism for data transfer, instead providing completely separate methods, the user is unable to discern the difference between copying data between the various portions of one application and copying data between applications. Short-circuiting the data transfer request from one portion of the application to another eliminates the communication overhead when the transfer is within a single address space.

The ICCCM provides a large amount of mechanism built on top of the basic selection system; multiple targets, incremental selections, and a wide variety of standard data formats. Using some of these mechanisms, an application should be able to perform many of the editing functions normally reserved to reside within a single application transparently between unrelated applications. By negotiating the datatype of the transfer, two cooperating applications can transmit arbitrary data, while still allowing transfer of the same information in a standard format to other clients.

3. The raw bits. X Protocol encodings and semantics

This section will describe the wire format of the various requests which are needed when using selections.

There are three X requests which relate directly to selections: SetSelectionOwner, GetSelectionOwner and ConvertSelection. There are also three events: SelectionClear, SelectionRequest and SelectionNotify. SendEvent is used to generate some of these events when necessary.

3.1. SetSelectionOwner

selection: **ATOM**
owner: **WINDOW or** `None`
time: **TIMESTAMP or** `CurrentTime`
Errors: `Atom`, `Window`

This request asserts ownership of a selection. The selection is only owned if the given TIMESTAMP is between the TIMESTAMP given the last time the selection ownership was set and the current server time. The *owner* field specifies the window which is somehow related to the selection. Typically it refers to the window in which the selection is displayed; if the window is not displayed, it may have some other meaning.

The ICCCM demands that the window be owned by the selection client. This makes sense; there is no other way for the selection requestor to identify the client which owns the selection. When this window is destroyed, the selection will automatically be lost.

When a client wants to volunteerly surrender a selection, it simply sets the owner window to `None` for this request.

The current owner of the selection discovers that it has lost the selection by receiving a SelectionClear event; if the current "owner" window is `None`, then no event will be generated.

An important note here: there is no indication of failure from this request if the timestamp given is not within the specified bounds. Therefore, the client must use GetSelectionOwner immediately afterwards to verify that the selection is indeed owned.

3.2. SelectionClear event

owner: **WINDOW**
selection: **ATOM**
time: **TIMESTAMP**

The fields in this event are the values stored when the SetSelectionOwner request which this client sent to own this selection was processed. If the time given in the SetSelectionOwner request was `CurrentTime`, then the time at which the SetSelectionOwner request was processed will be sent.

3.3. GetSelectionOwner

selection: **ATOM**
Errors: `Atom`

This request simply returns the current owner window of the selection. It would be very nice if the timestamp was also included in this reply so that clients could discover if the selection had been updated. Unfortunately, this is not the case, making it effectively impossible to discover when the selection has changed.

3.4. ConvertSelection

selection: **ATOM**
target: **ATOM**
property: **ATOM or** `None`
requestor: **WINDOW**
time: **TIMESTAMP or** `CurrentTime`
Errors: `Atom`, `Window`

This is the request which the selection requestor uses to direct an event at the current selection holder. The additional fields are used by the ICCCM to transmit information associated with the selection.

One of two events will be generated by this client, depending on whether this selection is owned by any client. If it is, that owning client receives a `SelectionRequest` event. Otherwise the requestor receives a `SelectionNotify` event with property `None` (see below).

Because the selection owner is determined when the request is processed, it is guaranteed to be received by the owner client before the owner client could potentially receive a SelectionClear event. This avoids the potential troubles which would be caused if the requestor discovered the owner, the ownership was transfered to another client and the requestor then directed a request at the previous owner. The problem which remains is that the client may have disowned the selection voluntarily before receiving the SelectionRequest event; the ICCCM allows clients to respond to requests in the past in whatever way seems reasonable to the owner.

The ICCCM semantics of this request are simple: the requestor window, which must be owned by the requesting client, will be used to aid transfer the data in two ways. First the data are transfered via a property stored on this window. The requestor generates the property name itself, to avoid potential conflicts if it allowed the owner to do so. Second, the Window will be used as the target of a SendEvent call by the owner, sending a SelectionNotify event to indicate when the data are ready.

The *target* atom is used by the ICCCM to select the type of the data to be transfered. There are several pre-defined targets, one of which is a request for a list of types which the owner supports for this selection. The ICCCM selection below describes this in more detail.

3.5. SelectionRequest event

owner:	**WINDOW**
selection:	**ATOM**
target:	**ATOM**
property:	**ATOM or** `None`
requestor:	**WINDOW**
time:	**TIMESTAMP or** `CurrentTime`

Most of the arguments in this event are copied directly from the ConvertSelection request which generated it. The *owner* field is filled in from the current state of the selection.

In a divergence from its usual role as providing only mechanism, the protocol describes the expected policy for using this information, instead of defering that to the ICCCM. When the owner receives this event, it is expected to place the requested datatype in the specified on the specified window. Then it is expected to generate a `SelectionNotify` event directed at the owner of the *requestor* window by setting the eventmask in that request to 0 (see below).

There is a proposed policy of having the selection owner determine which property to use when `None` is specified. This deprecated by the ICCCM as dangerous; any choice may conflict with an existing property.

3.6. SelectionNotify events

requestor:	**WINDOW**
selection:	**ATOM**
target:	**ATOM**
property:	**ATOM or** `None`
time:	**TIMESTAMP or** `CurrentTime`

When a ConvertSelection request is executed, the client can always expect one of these events in return; either generated directly by the server in the case where the requested selection is unowned, or generated by the owner when there is one. When generated by the selection owner, a *property* of `None` indicates that the selection conversion failed, most likely because the requested *target* was not acceptable by the client.

3.7. GetProperty

window:	**WINDOW**
property:	**ATOM**
type::	**ATOM or** `AnyPropertyType`
long-offset, long-length:	**CARD32**
delete:	**BOOL**
→	
type:	**ATOM or** `None`
format:	**{ 0, 8, 16, 32 }**
bytes-after:	**CARD32**
value:	**LISTofINT8 or LISTofINT16 or LISTofINT32**

The requestor receives the selection data by fetching the value of the property specified in the *SelectionNotify* event. Because the ICCCM allows the owner to use whatever type it chooses, the requestor should use `AnyPropertyType` in this request. Otherwise the server will not return the data. As the requestor is responsible for cleaning up after the selection data has been transfered, for simple selections *delete* should probably be set to TRUE.

Note that this request attempts to confuse you by having a the field which corresponds to the selection target called *type*.

4. ICCCM extensions to the basic structure

Upon these raw bits, the ICCCM has added quite a lot of functionality.

4.1. Selection names

There are only three selections which the ICCCM defines; PRIMARY, SECONDARY and CLIPBOARD. The first two are pretty straight forward, while CLIPBOARD has some additional semantics and references to a separate client, the clipboard manager.

PRIMARY is sort of the default selection; it should be used unless the clients have a good reason for using another. This is the selection which is supposed to be represented graphically on the display; for example a text editor would assert this selection whenever the user selected a range of text and would highlight the region in inverse video or some other means.

SECONDARY is sort of a catch all for things which don't fit under the PRIMARY umbrella. This selection should be used when the application wishes to leave the contents of the PRIMARY selection alone, or in environments where the action to be applied to the selections need two arguments, instead of one, for example, exchanging two pieces of data. In this later role, we'll see the special target INSERT_SELECTION quite useful.

The CLIPBOARD selection is designed to provide a cut/copy/paste system by allowing a client to hold a selection which is needn't be displayed on the screen, and which should last longer than the lifetime of a user activity. The ICCCM also describes a mechanism for transfering the contents of this selection into another process, allowing the original owner the freedom to forget the contents, or to simply disconnect from the server. This resolves the problem of how to handle long term storage; that additional client can be as long-lived as the X server itself.

The MIT client **xclipboard** is designed to provide this service. It also provides a list of previous CLIPBOARD contents, keeping each in a separate buffer and allowing the user to choose which will be returned when the CLIPBOARD selection is requested.

Having a separate client has pluses and minuses. Allowing the user to see the contents of the CLIPBOARD in a separate window provides visual feedback that the selection has been made, while keeping a kill-ring of previous CLIPBOARD contents provides an emacs-style history, except with global scope instead of being limited to a single client. Naturally, the original client can discontinue storage of the selection contents, and even disconnect from the server, preserving the contents of the CLIPBOARD independent of the state of individual clients.

On the minus side, the separate clipboard client may not support the datatype that the future selection requestor would like. It also eliminates a nice feature of the other selection mechanisms by insisting that the data be transfered, even if the contents will never be requested. Because of that, it is expected that CLIPBOARD selections will be less frequent than either PRIMARY or SECONDARY. Each client should be aware of this and design the interfaces appropriately. Each client should also be aware that a separate clipboard client may not exist, and should be prepared to hold the clipboard contents itself.

4.2. Selection targets

If you look over this section in the ICCCM you'll see an entire page of predefined selection targets. Most of them are pretty straight forward, but a few require some additional clarification.

4.2.1. TARGETS

This returns a list of valid target atoms. This is so the requestor can discover which targets are supported before asking for a specific target. This can be useful when the requestor supports multiple representations, and would like to get the most useful one. The list of targets needn't be checked when asking for a very simple one; the MIT clients which request STRING data never bother checking whether the owner supports it.

4.2.2. COMPOUND_TEXT

The ICCCM does not include this target, which will become more important in the future as the efforts toward internationalization progress. This target specifies the X consortium standard multi-lingual text encoding mechanism Compund Text. This is an extension of the regular ISO Latin-1 set; extended with escape sequences which embed other encoding sequences, including multi-byte characters. The MIT X clients support this target by simply ignoring the portions of the selection which are not Latin-1. I don't recommend this approach, but it is probably better than nothing.

4.2.3. MULTIPLE

This is not a true target, but a pointer to a list of desired targets. When this target is used, a list of pairs of (target,property) is placed in the property which will eventually contain the selection data. This means that the ConvertSelection request must contain a property for this purpose (otherwise the selection owner would not be able to discover the desired targets). The selection owner places the results of each target on the associated property in the order in which the targets were specified. This allows the following selections to have well-defined meaning.

4.2.4. DELETE
INSERT_SELECTION
INSERT_PROPERTY

These exist to provide semantics beyond the simple transfer of data. DELETE is supposed to cause the owner client to delete the representation of the selection contents from the source data object. This provides "cut" semantics.

INSERT_SELECTION/INSERT_PROPERTY, when combinded with DELETE provide the ability to swap the contents of selections. For example, to swap PRIMARY and SECONDARY, the SECONDARY owner would simply send a ConvertSelection using MULTIPLE as the target and PRIMARY as the selection, for the list of (target, property), use (TEXT,TEXT) (DELETE,DELETE) (INSERT_SELECTION,SECONDARY). This causes the selection owner to transfer the TEXT data, delete the TEXT data and then request the SECONDARY contents which it places in the same place that the TEXT contents used to reside. Now the two clients will switch roles as the original PRIMARY owner will request the SECONDARY contents.

4.3. Large selections

The X server is expected to have limited storage capacity for selection data, in some cases the size of the selection data may exceed the servers ability to store it. Also, it is generally considered impolite to place huge demands on server storage simply to provide inter-client communication.

To fix this problem, the ICCCM defines a way of transmitting the data incrementally, allowing the data to stream through the server in smaller chunks. For those using the X toolkit, this mechanism is handled behind the scenes and can safely be ignored. However, if you are adamant about getting in on helping the toolkit, you can do it yourself as the toolkit provides the appropriate hooks.

If the size of the selection is large, the owner responds to the SelectionRequest event by placing a single integer in the property specified in the event and generating a SelectionNotify event. Instead of returning the requested target, the owner returns INCR. When the requestor receives a event with this target, it knows that the transfer will be in pieces, rather than all at once.

5. Synchronization and Timestamps

When a client asserts ownership of a selection, using SetSelectionOwner, it includes in the request a Timestamp which is usually the time at which the user action which guided this process occured. If the current selection owner specified a later timestamp, the SetSelectionOwner request fails and when the client checks the ownership of the selection it will notice this.

This allows the ownership of the selection to follow the actions of the user, rather than be dependent on the vagaries of the network. As an example where this makes a difference, suppose a server has two clients, cleint A which responds very quickly to user demands and client B which takes quite a long time. The sequence of events would be:

Server	Client A	Client B
User invokes selection action for client B		
		Client B receives the event
User invokes selection action for client A		
	Client A receives the event	
	Client A responds to its event, calling SetSelectionOwner	
Selection owned by client A		
		Client B responds to its event, calling SetSelectionOwner
Timestamp from B < A preventing ownership by client B		

If the server performed the actions without regard to the timestamps inside the SetSelectionOwner requests, the selection would end up owned by client B, rather than owned by client A as the user had (presumably) intended. This is one of the many instances in which the protocol uses Timestamps to provide a means of sequencing actions according to the sequence of user actions which invoked them, rather than according to the sequence of protocol requests that those actions generate.

6. Selections and the X Toolkit

After the ICCCM was agreed upon and standarized, the X toolkit intrinsics were extended to include support for the new selection conventions, along with many other new features. Release 4 represents these changes, intrinsics from before this era may or may not have the support described below. Along with the Xt support, some Xmu routines also exist to handle some of the grunge.

For the selection owner, Xt hides the details of MULTIPLE targets, and has support for either helping or completely hiding the details of incremental transfer. Xt also provides notification when the selection is lost. For the selection requestor, Xt provides support for MULTIPLE targets and hides the details of incremental transfers.

Xt also provides for local transfer of selection contents within a single application. This allows the application to be coded without regard to where the selection contents will be transfered. Xt automatically short-circuits the wire protocol, eliminating all network data transfer.

The entire selection system uses callbacks to avoid nested event loops. Unfortunately, this frequently causes recursive invocation of the selection code, as a callback routine attempts to perform some other selection activity. For example, the CLIPBOARD client, xclipboard, is supposed to request the contents of the CLIPBOARD selection whenever the selection is lost. Therefore, in the SelectionLost callback procedure, it calls XtGetSelectionValue. This exposed a bug in the R3 intrinsics. Also, in implementing the sample application for this paper, a bug in the R4 intrinsics was found when the ConvertSelection callback deleted the selection. This caused the SelectionDone callback to be invoked with a NULL widget. A suitable work-around for this later bug can be found in the source code.

6.1.

Boolean XtOwnSelection

```
Widget          widget;
Atom            selection;
```

Time	**time;**
XtConvertSelectionProc	**convert;**
XtLoseSelectionProc	**lose;**
XtSelectionDoneProc	**done;**

Boolean XtOwnSelectionIncremental

Widget	**widget;**
Atom	**selection;**
Time	**time;**
XtConvertSelectionIncrProc	**convert;**
XtLoseSelectionIncrProc	**lose;**
XtSelectionDoneIncrProc	**done;**
XtCancelConvertSelectionProc	**cancel;**
XtPointer	**closure;**

These two provide a simple interface to aquiring a selection. They perform the XSetSelectionOwner and XGetSelectionOwner requests necessary to aquire and verify selection ownership, returning False if the selection could not be aquired. It is unfortunate that the client must specify at this time, rather than when the selection is converted, whether the selection conversion data will be "owned" by the toolkit or not ("owned" means which portion of the application is responsible for cleanup when the selection transfer is complete.) In many cases, the client would like to own the bulk data transfer data, while allocating storage for the smaller targets and allowing Xt to free them when done. It is also unfortunate that the application must choose up front whether it will support Incremental selections itself, or let the toolkit perform them.

6.2.

Boolean (*XtConvertSelectionProc)

Widget	**widget;**	
Atom	***selection**	
Atom	***target;**	
Atom	***type;**	**/* RETURN */**
XtPointer	***value;**	**/* RETURN */**
unsigned long	***length;**	**/* RETURN */**
int	***format;**	**/* RETURN */**

Called when the value of the selection has been requested, this routine returns True if the conversion succeeds, and False otherwise. The reason all of the parameters are pointers is so that this routine can be written in Fortran. Probably the lamest requirement for C toolkit I can think of. The return type needn't match the requested type; it should instead reflect the actual data format returned. Choosing another datatype arbitrarily is probably a bad idea; the requesting client will potentially not understand the replacement type. One case where it is reasonable is where the requested type is TEXT - as the ICCCM does not specify an encoding for that type, returning either STRING, indicating ISO 8859.1 encoding, or COMPOUND_TEXT, indicating the X Compound Text encoding, is a good idea.

6.3.

void (*XtLoseSelectionProc)

Widget	**widget;**
Atom	***selection;**

This is called when the widget has lost the specified selection. Note that this is not a request for the widget to lose the selection, rather simple notification of an accomplished fact.

6.4.

void (*XtSelectionDoneProc)
Widget widget;
Atom *selection, *target;

When some transfer has completed from the specified selection, this routine is called. If, in the XtOwnSelection/XtOwnSelectionIncremental call, a Null function pointer is passed, then the Intrinsics will automatically call XtFree on the selection data returned from the ConvertSelection proceedure. No sort of identification of which selection request has finished is passed in this routine. If more than one selection request is going on at a time, the client will have to be very careful about freeing data associated with an individual selection request in this procedure, lest it free the data associated with the wrong request. A suitable way around this difficulty would be to allocate data per target, instead of per request, reference counting the number of requests for a particular target, and freeing the data when all ongoing requests for a particular target complete. Alternatively, the client could simply hold all data for all requests until all requests are finished.

6.5.

void XtDisownSelection
Widget widget;
Atom selection;
Time time;

This routine disowned the specified selection. The LoseSelection proceedure is called to notify the current owner. This routine does nothing if the selection is not owned by this client. The *time* argument is used in the SetSelectionOwner request; ICCCM requires this value to be the value used in the XtOwnSelection call. I don’t know why the toolkit doesn’t simply store that value for use in this request.

6.6.

XSelectionRequestEvent *XtGetSelectionRequest
Widget widget;
Atom selection;
XtRequestId request_id;

The intrinsics save the event which caused the ConvertSelection proceedure to be called for situations which require it. When using the Xt incremental selection mechanism, the request_id argument passed to the ConvertSelectionIncremental procedure must be passed back to this routine. XtGetSelectionRequest can only be called while the ConvertSelection procedure is active.

6.7.

void XtGetSelectionValue
Widget widget;
Atom selection;
Atom taget;
XtSelctionCallbackProc callback;
XtPointer client_data;
Time time;

XtGetSelectionValueIncremental
Widget widget;
Atom selection;
Atom taget;
XtSelctionCallbackProc callback;

XtPointer	**client_data;**
Time	**time;**

Either of these procedures can be used to get the value of a selection. Unless a partially successful incremental selection transfer is interesting, the second routine is not very useful; it simply provides a way for the application to get involved in the incremental process. In either case, the callback function is called with the selection value, piecewise for XtGetSelectionValueIncremental, and all at once for XtGetSelectionValue.

6.8.

XtGetSelectionValues

Widget	**widget;**
Atom	**selection**
Atom	***targets**
int	**count;**
XtSelectionCallbackProc	**callback;**
XtPointer	**client_data;**
Time	**time;**

XtGetSelectionValuesIncremental

Widget	**widget;**
Atom	**selection**
Atom	***targets**
int	**count;**
XtSelectionCallbackProc	**callback;**
XtPointer	**client_data;**
Time	**time;**

These two use the MULTIPLE target ICCCM stuff to get more than one target from a single selection at a time. The callback is called for each successful target conversion.

6.9. XtAppGetSelectionTimeout, XtGetSelectionTimeout XtAppSetSelectionTimeout, XtSetSelectionTimeout

These set/get the application resource which controls the length of time which Xt will wait for selection data. See the section on Selection Timeouts below.

7. Some useful Xmu routines

The Xmu library contains a few useful tools for manipulating selections. While this library is not a standard, it is freely available on the MIT release.

7.1. #include <Xmu/Atoms.h> AtomPtr XmuMakeAtom(char *name) Atom XmuInternAtom(Display *d, AtomPtr *atom_ptr)

Because many of the atoms used to refer to selections are not a part of the built-in set, these two routines (along with others, see the Xmu library documentation) provide a convientent interface. Non built-in atoms are numbered by the server, instead of by clients. This ensures that all atoms of the same name use the same numeric ID, unfortunately, this also means that they will have different numeric representations on different displays.

These two clever routines carfully keep all of this straight. The first routine is used to build an opaque structure which is used to pass to the second each time the atom is referenced:

```
AtomPtr      _XA_FOO;
Atom Foo;

_XA_FOO = XmuMakeAtom ("FOO");
Foo = XmuInternAtom (display, _XA_FOO);
```

Because of the local cache of information, simply using the XmuInternAtom routine every time the name is referenced is not an unreasonable thing to do; by creating a CPP macro which does this somewhat transparently, it even becomes easy:

```
extern AtomPtr          _XA_FOO;
#define XA_FOO(d)       XmuInternAtom(d, _XA_FOO)
```

This naming style is a convention used by these routines; it follows the Xlib convention for naming the built-in atoms which is to prepend the name of the atom with "XA_".

For each of the atoms defined in the ICCCM, this has already been done

XA_ATOM_PAIR, XA_CHARACTER_POSITION, XA_CLASS, XA_CLIENT_WINDOW, XA_CLIPBOARD, XA_COMPOUND_TEXT, XA_DECNET_ADDRESS, XA_DELETE, XA_FILENAME, XA_HOSTNAME, XA_IP_ADDRESS, XA_LENGTH, XA_LIST_LENGTH, XA_NAME, XA_NET_ADDRESS, XA_NULL, XA_OWNER_OS, XA_SPAN, XA_TARGETS, XA_TEXT, XA_TIMESTAMP, XA_USER.

To use these, simply pass them a display pointer. Appropriate initialization has been set up so that these special atoms needn't be gotten from XmuMakeAtom.

7.2.

Boolean XmuConvertStandardSelection

Widget		
Time	**time;**	
Atom	***selection, *target;**	
Atom	***type;**	**/* RETURN */**
XtPointer	***value;**	**/* RETURN */**
unsigned long	***length;**	**/* RETURN */**
int	***format;**	**/* RETURN */**

There are quite a few targets which don't actually involve the data selected; for these, this routine has been written to take care of the details. It handles: TIMESTAMP, HOSTNAME, IP_ADDRESS, DECNET_ADDRESS, USER, CLASS, NAME, CLIENT_WINDOW, OWNER_OS, TARGETS

The ICCCM requires that the value returned for the **TIMESTAMP** target be the value used in accquiring the selection, therefore the value passed as **time** to this function should contain that value. This allows clients to discover the time associated with a selection, which is not obtainable directly through the protocol. **CLASS** and **CLIENT_WINDOW** are computed from the widget argument, while the remaining targets are determined from the execution environment in an operating system-dependent manner. Using this function allows the application to be ignorant of these host system details.

The TARGETS which this routine supports should be added to the other TARGETS which the application supports, by calling this routine with the target set to TARGETS and adding that result to the targets supported by the application ConvertSelection routine. This routine always allocates data for the target, so if you specify a Done proc in the XtOwnSelection call, that procedure will have to be able to free the data allocated using XtFree. Therefore you'll have to save away the pointer returned, as the Done proc does not receive that.

8. Selection Timeout

Because there is no negative indication of selection transfer, the requesting client simply waits for a while and then gives up, assuming that some bug exists in the owner, or that the owner has disconnected from the server after the SelectionRequest was processed. The default value which Xt uses for this purpose is 5 seconds, as a compromise between potential network latencies and user frustration (specified in the application resource **selectionTimeout**, class **SelectionTimeout** in milliseconds). When the timeout expires, Xt gives up on the selection transfer.

Another common reason for having the selection response take so long is that the owner is sitting inside the ConvertSelection callback computing the value before returning it for transmission. Because of the simple Xt interface, the entire value of the selection must be computed before any of the transfer can commence. When the selection is large, and the data must be reformatted, this can take quite a long time. The typical example of this is the MIT Athena Text widget which stores the text information in a complicated data structure. When asked to convert the selection, the Text widget copies each line of text separately into an allocated buffer. If the amount is large, this can take quite some time.

To avoid this problem, the application must perform the incremental operation itself, using XtOwnSelectionIncremental, so that it can send the first chunk of data as soon as possible, if the conversion of the entire value is expensive. As there is no way to specify this when the selection is requested, it must be done when the selection is owned. Alternatively, the selection data could be computed in the expected format before the selection was owned. I only recommend this later solution when the data is stored internally in this format.

9. The sample application

While writing this paper, I developed a sample text editing widget which uses selections extensively; avoiding all other mechanisms of retaining data. This editor performs simple cut/copy/paste operations reminiscent of the Smalltalk or Macintosh systems (although smalltalk used a menu for the editing functions, I have used the keyboard for simplicity).

I attempted to elide uninteresting code, and yet provide a complete working example. Even with my restraint the application contains 1200 lines of C code, a large portion of which is classic Xt boilerplate required for any widget. I am not presenting this code as the only way in which selections should be used, in fact I'm convinced that some of the decisions made were not the correct ones. It does demonstrate that selections are sufficient for many common editing tasks, going beyond the example provided by the Athena text widget, which I feel has not succesfully integrated selections into its structure.

10. Debugging with selections

When writing applications which use selections, there are only a few really helpful suggestions. First, if you plan to program using the Xt intrinsics, get Saber C. This C interpreter finds many bugs which regular C compilers simply cannot find. If you have source to the toolkit as well, you'll get even more checking by loading the source version of appropriate portions of the toolkit.

Second, get xscope. Xscope is supplied with the R4 distribution as a contributed client and provides a trace of the protocol requests and events related to particular clients. When attempting to debug the fine details of almost any selection-based program, the ability to determine that the correct events and requests are being generated is invaluable. Even when using the "assistance" provided by Xt, getting the correct protocol is heavily dependent on the correctness of the application code.

Finally, when debugging, make sure you run all of the applications with long enough selectionTimeouts to avoid getting caught by one when you are stopped at a breakpoint in the convertSelection callback. Fortunately, there is a default command-line option for specifying the selectionTimeout; -selectionTimeout <timeout>. I use -selectionTimeout 60000000 (1000 minutes).

11. Summary

I nearly wrote "Conclusion" above this section, but realized that no real conclusions can be drawn about selections. They are the chief mechanism used for inter-client data communication, so you'll end up using them, even if you don't really want to. There are a few hidden pitfalls in the design of selection-related software, and I hope I’ve uncovered most of them in this discussion.

Naturally, as with most other X software, writing selection related code is not an easy task. The Xt intrinsics do help with some of the low-level work, and provide a reasonable framework for integrating into an application, but they also provide traps for the unwary. As with most Xt programming, using the selection helper functions does not mean that your program will be easy to write, or even short, but they do help you structure the application to avoid the most obvious pitfalls. The ICCCM strongly suggests that the selection method is the preferred mechanism for communication, and I echo this suggestion, even if it requires more effort in the initial implementation.