XEmbed Protocol Specification

Mathias Ettrich

<ettrich@trolltech.com>

Owen Taylor

<otaylor@redhat.com>

Version 0.5

Table of Contents

Overview

XEmbed is a protocol that uses basic X mechanisms such as client messages and reparenting windows to provide embedding of a control from one application into another application. Some of the goals of the XEmbed design are:

1. Support for out-of process controls, written in any toolkit or even plain Xlib.

2. Support for in-process-controls when mixing different toolkits in one process.

3. Smooth integration of the embedding application and embedded client in areas such as input device handling and visual feedback.

4. Easy implementation. A full implementation supporting all details correctly may require minor toolkit modifications, but it should be possible to get basic functionality going in less than 1000 lines of code.

Goal 1 is the most urgent one. A embedding specification allows developers to write applets for whatever desktop the user is using in whatever toolkit they prefer. Goal 2 is more of something to keep in mind than a immediate requirement. While there are other ways to mix two or more toolkits, using XEmbed might be the easiest and thus most comfortable way. Goal 3 describes the targeted level of integration. The users should not necessarily notice that they work with embedded controls; devices like the keyboard and the mouse should work as expected, inactive windows should look like they are inactive, and so forth. The level of integration may, however, be limited by goal 4. In order for the protocol to be successful, it's crucial to get implementations for the most important toolkits. Thus, the implementation should not require too much coding and no or only few modifications to the toolkit's kernel.

At the time of writing, an implementation of XEmbed is included in GTK+-2.0 that mostly conforms to this version of the specification. The main area of divergence is in the area of accelerators, where a simpler scheme is implemented than the XEMBED_REGISTER_ACCELERATOR, XEMBED_UNREGISTER_ACCELERATOR accelerator scheme described here. The KDE libraries (libkdeui) include QXEmbed, a mostly-complete implementation for Qt of an earlier version of the protocol.

Definitions

Active

> A toplevel window is *active* if it currently is receiving keyboard events. (The window or a descendant has the X keyboard focus.) A widget within the toplevel is active if the toplevel is active, regardless of whether that

widget has the input focus within the toplevel.

Client

In an embedding situation, the *client* is the window that is embedded into an embedder. Sometimes also called a plug. (Note that the usage here should not be confused with the typical X usage of "client" to mean an application connecting to the X server. That is always referred to as an application in this document)

[Should we replace client by some other term in this document to avoid the confusion?]

Embedder

In an embedding situation, the *embedder* is the graphical location that embeds an external client. Sometimes also called a socket or site.

Focused

A widget is *focused* if it receives keyboard events within its toplevel. This is without regard to whether the toplevel is active, and has nothing to do with the X keyboard focus.

Rationale and discussion

The basis for handling embedding is that the embedder acts like a "window manager" for the client. (The window management protocol is defined in the X Inter-Client Communications Manual or ICCCM). The embedder selects with SubstructureRedirectMask on its window so that it can intercept, and then the client window is reparented (using `XReparentWindow()`) as a child of the embedder window. Because of the substructure redirect, the embedder is able to intercept calls to move or resize the client window, and handle them as appropriate to the location in the embedding application. (Map requests are also redirected, but XEmbed actually handles map requests separately... see the description of the XEMBED_MAPPED flag.)

The window management protocol is sufficient to handle the basics of visual

embedding, but has deficiencies in other areas that prevent it from providing natural integration between toolkits. These areas include:

window activation state
keyboard focus
tab focus chain
keyboard short cuts / accelerators
modality
drag and drop (XDND)

The XEmbed protocol is mainly concerned with communicating additional information between embedder and client to handle these areas. Communication in XEmbed is done by forwarding slightly modified XEvents using `XSendEvent()`, by sending special XEmbed messages, and by setting X properties. In addition, standard ICCCM features like WMNormalHints are used where appropriate.

The next sections explain why these problems occur with the simple "window management" approach and how XEmbed solves them.

Window activation state

A widget has to know the activation state of its toplevel window. This enables input widgets like a line editor, to display a blinking cursor only when the user can actually type into it. In addition, certain GUI styles choose to display inactive windows differently, typically with a lighter and less contrasting color palette.

Unfortunately, there are no such messages like WindowActivate or WindowDeactivate in the X protocol. Instead, a window knows that it is active when it receives keyboard focus (FocusIn event with certain modes) or looses it (FocusOut event with certain modes). This applies to embedded child windows only, when the mouse pointer points onto one of the child's subwindows in the very moment the window manager puts the X focus on the toplevel window. For that reason, XEmbed requires the embedders to pass XEMBED_WINDOW_ACTIVATE and XEMBED_WINDOW_DEACTIVATE messages to their respective clients whenever they get or loose X keyboard

Keyboard focus

The delivery of keyboard events in X is designed in a way that does not correspond to the typical operation of modern toolkits; instead it seems designed to allow things to allow things to work without either a window manager or a focus handling in the toolkit. Typically, key events are sent to the window which has the X input focus (set with XSetInputFocus()). However, if the mouse pointer is inside that focus window, the event is sent to the subwindow of the focus window that is under the moues pointer. In modern toolkits, the X input focus is typically left on the toplevel window and a separate logical input focus is implemented within the toolkit. The toolkit ignores the window that the key event is actually sent to (which might be a scrollbar or other random widget within the toplevel, depending on where the mouse pointer is), and distributes key events to widget with the logical input focus.

So, for standard operation, the behavior where key events are sent to the window with the mouse pointer is simply ignored. But with embedded windows, it causes problems, since, if the mouse pointer is within the embedded window, the outer toolkit doesn't see any key events, even if the logical keyboard focus is elsewhere within the outer toolkits toplevel window.

Previous embedding techniques therefore required clients to forward any key event they receive (KeyPress and KeyRelease) to their respective embedder. In order to support multiple levels of embedding, events that stem from a SendEvent request had to be forwarded as well. While this is a possible solution, it adds both race conditions and inefficiency.

The solution proposed by XEmbed is is to beat X11 with its own weapons: The topmost toolkit is *required* to keep the X input focus on one of its own windows without any embedded children. Keeping the focus on such a window ensures that key events are always delivered to the outer toolkit and thus can be forwarded easily to any embedded window. This also makes it possible to use this part of XEmbed with clients that do not support the protocol at all, without breaking keyboard input for the embedding application.

In detail, the topmost embedder creates a not-visible X Window to hold the focus, the focus proxy. (It might be a 1x1 child window of toplevel located at -1,-1.) Since the focus proxy isn't an ancestor of the client window, the X focus can never move into the client window because of the mouse pointer location. In other

words, whenever the outer window is activated (receives the X input focus), it has to put the X focus on the FocusProxy by calling `XSetInputFocus()`.

The trouble with this is, that you should not use `XSetInputFocus()` without a proper time stamp from the Server, to avoid race conditions. Unfortunately, the FocusIn event does not carry a timestamp. The solution to this is, to ask the window manager for the WM_TAKE_FOCUS window protocol. Thus, whenever the window is activated, it will receive a WM_PROTOCOLS client message with data.l[0] being WM_TAKE_FOCUS and data.l[1] being a proper timestamp. This timestamp can be used safely for the call to `XSetInputFocus()`.

If an embedder widget gets the logical input focus, it sends an XEMBED_FOCUS_IN message to its client. The client that receives this messages knows that its logical focus is now also the logical focus of the application window and will react accordingly. If its logical focus lies on the line editor control mentioned above, and the window is active, the editor will show a blinking cursor after processing this message.

In a similar fashion, if the embedder looses focus, it sends an XEMBED_FOCUS_OUT message.

Tab focus chain

X does not have a concept of a tab focus chain, it is up to the toolkit or the application to implement it. Since the concept is standard among almost all toolkits, XEmbed supports it. An XEmbed client integrates perfectly in the embedder's tab focus chain, i.e. the user can tab onto the client, through all its widgets and back to the outer world without noticing that they traversed an external window.

As explained in the previous section, an embedder sends an XEMBED_FOCUS_IN message to its client when it gets focus. The detail code of this message is per default 0, that is, XEMBED_FOCUS_CURRENT. It indicates that the clients keeps its own logical focus where it was. To support tabbing, XEmbed provides two more detail codes, namely XEMBED_FOCUS_FIRST and XEMBED_FOCUS_LAST, that indicate that the client should move its focus to the beginning or end of the focus chain.

When the user tabs to the very end of a client's tab chain, the client follows the request (i.e. it puts its logical focus back to the beginning its tab chain) and sends an XEMBED_FOCUS_NEXT message to the embedder. If the embedder has siblings that accept tab focus, it will do a virtual tab forward. As a result, it will loose focus itself and consequently send an XEMBED_FOCUS_OUT message to the client. As expected, the client's line edit control from the previous example will stop blinking.

Backward tabbing is done exactly in the same manner, using the XEMBED_FOCUS_PREV message.

Keyboard short cuts / accelerators

XEmbed is designed in such a way, that keyboard events are received by the toplevel window, and then sent down the focus focus chain. Toolkits will usually check for shortcuts or accelerators before sending the event to the focus widget. If such a shortcut is defined, the respective action is taken rather than passing the event through to the focus widget. This means, accelerators in the outmost window always work properly, whereas accelerators defined inside an embedded client only work if that client actually has focus. XEmbed solves this problem with two messages, XEMBED_REGISTER_ACCELERATOR and XEMBED_UNREGISTER_ACCELERATOR. With XEMBED_REGISTER_ACCELERATOR, a client can reserve a certain key/modifier combination as shortcut or accelerator. The message is passed through to the topmost embedder, where the key combination is stored. An XEMBED_UNREGISTER_ACCELERATOR message releases the key again.

Modality

If an application window is shadowed by a modal dialog, no user input is supposed to get through. The XEmbed design ensures this for keyboard input, because the toplevel window knows about its modal state and will not pass key events through. Embedded clients thus inherit the modality from the topmost embedder. Mouse input, however, is sent directly to the embedded clients by the X-Server, unaffected by the modality of the application window. To give clients the possibility to behave correctly when being shadowed by a modal dialog, an embedder can choose to send an XEMBED_MODALITY_ON message to its

client when it becomes shadowed, and an XEMBED_MODALITY_OFF message when it leaves modality again. If the client contains embedders itself, those have to pass both messages through to their clients.

Drag and drop (XDND)

XDND drag-and-drop does not work with reparented external windows, since messages are exchanged with the toplevel window only. This is done for performance reasons. While it is cheap to get the window under the mouse pointer, it is very expensive to get a window under another window. Unfortunately, this is required quite often when dragging objects around, since the pointer may overlap the drag icon.

Solving the drag-and-drop problem, however, is quite easy, since the XDND protocol was carefully designed in a way that makes it possible to support embedded windows. Basically, the embedder has to operate as drag-and-drop proxy for the client. Any XDND messages like XdndEnter, Xdnd,Leave, etc. simply have to be passed through. A toolkit's XDND implementation has to take this situation in consideration.

Embedding life cycle

The protocol is started by the embedder. The window ID of the client window is passed (by unspecified means) to the embedding application, and the embedder calls `XReparentWindow()` to reparent the client window into the embedder window.

Implementations may choose to support an alternate method of beginning the protocol where the window ID of the embedder is passed to client application and the client creates a window within the embedder, or reparents an existing window into the embedder's window. Which method of starting XEmbed is used a matter up to higher level agreement and outside the scope of this specification.

In either case the client window must have a property called _XEMBED_INFO on it. This property has type _XEMBED_INFO and format 32. The contents of the property are:

Table 1. _XEMBED_INFO

Field	Type	Comments
version	CARD32	The protocol version
flags	CARD32	A bitfield of flags

The *version* field indicates the maximum version of the protocol that the client supports. The embedder should retrieve this field and set the data2 field of the XEMBED_EMBEDDED_NOTIFY to Min (*version*, *max version supported by embedder*). The version number corresponding to the current version of the protocol is 0. *[Should the version be defined as (Major << 16 | Minor) ?]*

The currently defined bit in the *flags* field is:

```
/* Flags for _XEMBED_INFO */
#define XEMBED_MAPPED                   (1 << 0)
```

XEMBED_MAPPED

> If set the client should be mapped. The embedder must track the flags field by selecting for PropertyNotify events on the client and map and unmap the client appropriately. (The embedder can leave the client unmapped when this bit is set, but should immediately unmap the client upon detecting that the bit has been unset.)
>
> *Rationale: the reason for using this bit rather than MapRequest events is so that the client can reliably control it's map state before the inception of the protocol without worry that the client window will become visible as a child of the root window.*

To support future expansion, all fields not currently defined must be set to zero. To add proprietary extensions to the XEMBED protocol, an application must use a separate property, rather than using unused bits in the struct field or extending the _XEMBED_INFO property.

At the start of the protocol, the embedder first sends an XEMBED_EMBEDDED_NOTIFY message, then sends XEMBED_FOCUS_IN,

XEMBED_WINDOW_ACTIVATE, and XEMBED_MODALITY_ON messages as necessary to synchronize the state of the client with that of the embedder. Before any of these messages received, the state of the client is:

Not focused
Not active
Modality off

If the embedder is geometry managed and can change its size, it should obey the client's WMNormalHints settings. Note that most toolkits will not have equivalents for all the hints in the WMNormalHints settings, clients must not assume that the requested hints will be obeyed exactly. The `width_inc`, `height_inc`, `min_aspect`, and `max_aspect` fields are examples of fields from WMNormalHints that are unlikely to be supported by embedders.

The protocol ends in one of three ways:

1. The embedder can unmap the client and reparent the client window to the root window. If the client receives an ReparentNotify event, it should check the `parent` field of the XReparentEvent structure. If this is the root window of the window's screen, then the protocol is finished and there is no further interaction. If it is a window other than the root window, then the protocol continues with the new parent acting as the embedder window.

2. The client can reparent its window out of the embedder window. If the embedder receives a ReparentNotify signal with the `window` field being the current client and the `parent` field being a different window, this indicates the end of the protocol.

   [GTK+ doesn't currently handle this; but it seems useful to allow the protocol to be ended in a non-destructive fashion from either end.]

3. The client can destroy its window.

Message Specifications

An XEmbed message is an X11 client message with message type "_XEMBED".

The format is 32, the first three data longs carry the toolkit's X time (l[0]), the message's major opcode (l[1]) and the message's detail code (l[2]). If no detail is required, the value passed has to be 0. The remaining two data longs (l[3] and l[4]) are reserved for data1 and data2. Unused bytes of the client message are set to 0. The event is sent to the target window with no event mask and propagation turned off.

The valid XEmbed messages are:

```
/* XEMBED messages */
#define XEMBED_EMBEDDED_NOTIFY          0
#define XEMBED_WINDOW_ACTIVATE          1
#define XEMBED_WINDOW_DEACTIVATE        2
#define XEMBED_REQUEST_FOCUS            3
#define XEMBED_FOCUS_IN                 4
#define XEMBED_FOCUS_OUT                5
#define XEMBED_FOCUS_NEXT               6
#define XEMBED_FOCUS_PREV               7
/* 8-9 were used for XEMBED_GRAB_KEY/XEMBED_UNGRAB_KEY */
#define XEMBED_MODALITY_ON              10
#define XEMBED_MODALITY_OFF             11
#define XEMBED_REGISTER_ACCELERATOR     12
#define XEMBED_UNREGISTER_ACCELERATOR   13
#define XEMBED_ACTIVATE_ACCELERATOR     14
```

A detail code is required for XEMBED_FOCUS_IN. The following values are valid:

```
/* Details for  XEMBED_FOCUS_IN: */
#define XEMBED_FOCUS_CURRENT            0
#define XEMBED_FOCUS_FIRST              1
#define XEMBED_FOCUS_LAST               2
```

XEMBED_EMBEDDED_NOTIFY

Sent from the embedder to the client on embedding, after reparenting and mapping the client's X window. A client that receives this messages knows that its window was embedded by an XEmbed site and not simply reparented by a window manager. To support toolkits that do not keep track of reparenting events, the message carries the embedder's window handle as data1:

Table 2. XEMBED_EMBEDDED_NOTIFY

data1	The embedder's window handle.
data2	The protocol version in use (see the description of _XEMBED_INFO).

XEMBED_WINDOW_ACTIVATE / XEMBED_WINDOW_DEACTIVATE

Sent from the embedder to the client when the window becomes active or inactive, i.e. when the window gets or looses the keyboard input focus. If the client contains embedders itself, those have to pass the message through to their clients.

Note that no XEMBED_FOCUS_IN or XEMBED_FOCUS_OUT messages should be sent when the toplevel window gains or loses focus. The XEMBED_FOCUS_IN and XEMBED_FOCUS_OUT messages refer only to focus *within* the toplevel window and are independent of toplevel activation state. This independence is necessary so that input focus within a toplevel can be moved programmatically when the toplevel doesn't have input focus.

[GTK+ is currently in violation of the preceding note, and sends FOCUS_IN and FOCUS_OUT only when the toplevel is active. See GNOME bug #67943]

Widgets within the client should typically be displayed with the focus only when the client both has focus and is active.

XEMBED_REQUEST_FOCUS

Sent from the client to the embedder when the client wants focus. The most common occasion is when the user clicks into one of the client's child widgets, for example a line editor, in order to type something in.

The message is passed along to the topmost embedder that eventually responds with a XEMBED_FOCUS_IN message. The focus in message is passed all the way back until it reaches the original focus requester. In the end, not only the original client has focus, but also all its ancestor embedders.

XEMBED_FOCUS_IN

Sent from the embedder to the client when it gets focus. The detail code determines, where the client shall move its own logical focus to. Three possibilities exist:

XEMBED_FOCUS_CURRENT

Normal activation, does not move the clients logical focus.

XEMBED_FOCUS_FIRST

Used when the user tabs onto a client. It indicates that the client should put its logical focus onto the widget that comes first in its own tab focus chain.

XEMBED_FOCUS_LIST

Used when the user tabs onto a client. It indicates that the client should put its logical focus onto the widget that comes first in its own tab focus chain.

XEMBED_FOCUS_OUT

Sent from the embedder to the client when it looses focus.

XEMBED_FOCUS_NEXT

Sent from the client to the embedder when it reaches the end of its logical tab chain after the user tabbed forward. If the embedder has siblings that accept tab focus, it will do a virtual tab forward. As a result, it will loose focus itself and consequently send an XEMBED_FOCUS_OUT message to the client

XEMBED_FOCUS_PREV

Sent from the client to the embedder when it reaches the beginning of its logical tab chain after the user tabbed backward. If the embedder has siblings that accept tab focus, it will do a virtual tab backward. As a result, it will loose focus itself and consequently send an XEMBED_FOCUS_OUT message to the client

XEMBED_REGISTER_ACCELERATOR /

XEMBED_UNREGISTER_ACCELERATOR

A client that needs to reserve a certain key/modifier combination as shortcut or accelerators, sends a XEMBED_REGISTER_ACCELERATOR message to its embedder. As long as the embedder itself is a child of a client, the accelerator will be propagated up to the toplevel.

Table 3. XEMBED_REGISTER_ACCELERATOR

detail	accelerator_id
data1	X key symbol
data2	bit field of modifier values

The accelerator_id is used to identify the accelerator when activating the accelerator. The reason for using an accelerator ID instead of identifying accelerators simply by key symbol and modifiers is to allow the correct handling of overloaded accelerators with embedded widgets. (An accelerator is overloaded if there multiple accelerators on the same key, usually because of accidental collisions.) When an overloaded accelerator is pressed repeatedly, the toplevel activates accelerators on that key in round-robin fashion. If this round-robin behavior is not supported by the embedding toolkit, picking an arbitrary accelerator for the key and activating it is acceptable. Well designed applications should avoid collisions in any case.

Note

Ordering the round-robin of conflicting accelerators in a predictable (geometric or in focus chain) order is desirable. This can be achieved if the toplevel sorts the conflicting accelerators as if they applied to the client instead of widgets within the client and then each client does the same sort on the subset of conflicting accelerators within it. To get this to work properly if there are conflicting accelerators within a client, say widget A and B both have the same mnemonic, then instead of registering one accelerator for widget A and one for widget B, the client should register two accelerators that corresponds to both A and B, and

then when XEMBED_ACTIVATE_ACCELERATOR is received for either accelerator, implement round robin between A and B with the correct sorting.

The modified bit field is a bitwise OR of values indicating various modifiers; these indicate logical accelerator keys rather than corresponding directly to the bits in the XKeyEvent state field.

```
/* Modifiers field for XEMBED_REGISTER_ACCELERATOR */
#define XEMBED_MODIFIER_SHIFT    (1 << 0)
#define XEMBED_MODIFIER_CONTROL  (1 << 1)
#define XEMBED_MODIFIER_ALT      (1 << 2)
#define XEMBED_MODIFIER_SUPER    (1 << 3)
#define XEMBED_MODIFIER_HYPER    (1 << 4)
```

(Meta is intentionally left out here because if you try to separate Alt and Meta, a large fraction of users will experience problems with their keyboard setups... there is no reliably standard of which one is the primary modifier key and on the Alt key.)

On activation, the topmost embedder will send XEMBED_ACTIVATE_ACCELERATOR to its client; if the accelerator was registered by an embedder inside that client, the embedder will send XEMBED_ACTIVATE_ACCELERATOR to its client and so forth.

Note that the assignment of ID's is private for each pair of client and embedder and when accelerators are being propagated through multiple client/embedder pairs, a different accelerator ID may be used for each pair.

The XEMBED_UNREGISTER_ACCELERATOR message releases the key combination again.

Table 4. XEMBED_UNREGISTER_ACCELERATOR

detail	integer ID passed to XEMBED_REGISTER_ACCELERATOR

Hint to implementors: It is the responsibility of the embedder to keep track of all forwarded accelerators and to remove them when the client window dies.

XEMBED_ACTIVATE_ACCELERATOR

The XEMBED_ACTIVATE_ACCELERATOR message is sent when a accelerator previously registered with XEMBED_REGISTER_ACCELERATOR is activated on the toplevel containing the embedder.

Table 5. XEMBED_ACTIVATE_ACCELERATOR

detail	integer ID passed when registering the accelerator
data1	flags.

The following bit is defined for the flags field; all other bits must be zero.

```
/* Flags for XEMBED_ACTIVATE_ACCELERATOR */
#define XEMBED_ACCELERATOR_OVERLOADED (1 << 0)
```

XEMBED_ACCELERATOR_OVERLOADED

> This flag indicates that multiple accelerators exist for the key combination within the toplevel. The toolkit may modify the behavior of the accelerator based on this value. For instance, if the accelerator is a mnemonic for a button, it might activate the the button immediately if the accelerator is not overloaded, but when overloaded, it would only focus the button.

XEMBED_MODALITY_ON / XEMBED_MODALITY_OFF

Sent from the embedder to the client when the window becomes shadowed by a modal dialog, or when it is released again. If the client contains embedders itself, those have to pass the message through to their clients. An embedded control should ignore mouse input while modality is active. Note that that keyboard input is blocked anyway by XEmbed, since the topmost embedder will not pass keyboard events through in modal state.

Techniques

Handling errors

Implementors of the XEmbed protocol should handle the other party disappearing at any point. For this reason X errors must be trapped when performing any operation with a window not created by the application. This is done by using `XSetErrorHandler()`. A sample implementation of trapping errors in C looks like:

```
#include <X11/Xlib.h>

static int trapped_error_code = 0;
static int (*old_error_handler) (Display *, XErrorEvent *);

static int
error_handler(Display     *display,
              XErrorEvent *error)
{
    trapped_error_code = error->error_code;
    return 0;
}

void
trap_errors(void)
{
    trapped_error_code = 0;
    old_error_handler = XSetErrorHandler(error_handler);
}

int
untrap_errors(void)
{
    XSetErrorHandler(old_error_handler);
    return trapped_error_code;
}
```

Forwarding X Events

An XEmbed embedder has to forward key-press and key-release events to its respective client.

Key events are forwarded by changing the event's window field to the window handle of the client and sending the modified message via `XSendEvent()` to the embedder, with no event mask and propagation turned off.

Note: XEmbed requires toolkits to handle key-events that come from a SendEvent request. That means, if somebody can access your X-Server, it's possible to fake keyboard input. Given that most toolkits accept sent key events today anyway and the X Server is typically protected through magic cookie authorization, this is not considered to be an issue. Applications with higher security requirements may choose not to use embedded components, though, and to filter out any events coming from `XSendEvent()`.

Given that Window client is the client's window handle, here is a piece of code of an imaginary event-loop in C that does the forwarding.

```
#include <X11/Xlib.h>

void handle_event(
        Display* dpy, /* display */
        XEvent* ev /* event */
){
    if ( ev->type == KeyPress || ev->type == KeyRelease ) {
            ev->xkey.window = client;
            trap_errors();
            XSendEvent( dpy, client, False, NoEventMask, ev );
            XSync( dpy, False );
            if (untrap_errors()) {
               /* Handle failure */
            }

            return;
    }
    ... /* normal event handling */
}
```

Sending XEmbed messages

Given that Time x_time contains the timestamp from the event currently being processed. (CurrentTime is generally the best choice if no event is being processed), here is a valid implementation in C of sending an XEMBED message:

```
#include <X11/Xlib.h>

void send_xembed_message(
     Display* dpy, /* display */
     Window w, /* receiver */
     long message, /* message opcode */
```

```
     long detail  /* message detail */
     long data1  /* message data 1 */
     long data2  /* message data 2 */
){
    XEvent ev;
    memset(&ev, 0, sizeof(ev));
    ev.xclient.type = ClientMessage;
    ev.xclient.window = w;
    ev.xclient.message_type = XInternAtom( dpy, "_XEMBED", False );
    ev.xclient.format = 32;
    ev.xclient.data.l[0] = x_time;
    ev.xclient.data.l[1] = message;
    ev.xclient.data.l[2] = detail;
    ev.xclient.data.l[3] = data1;
    ev.xclient.data.l[4] = data2;
    trap_errors();
    XSendEvent(dpy, w, False, NoEventMask, &ev);
    XSync(dpy, False);
    if (untrap_errors()) {
        /* Handle failure */
    }
}
```

Issues

Implementation of modality

The protocol could be simplified by removing the XEMBED_MODALITY_ON and XEMBED_MODALITY_OFF messages in favor of requiring the embedder to map an input-only window over it's child when it beings shadowed by a modal grab.

One possible reason for the current protocol is that a toolkit might want to have elements such as scrollbars that remain active even when grab shadowed. (I know of no toolkit that actually implements this.)

Clarify function of timestamps

The function of the timestamp arguments needs to be clarified, as well as the requirements for what should be passed in the field. The original draft of the specification contained the text about the determining the timestamp.

> The x time is to be updated whenever the toolkit receives an event from the server that carries a timestamp. XEmbed client messages qualify for that.
>
> Hint to implementors: Check that the xembed time stamp is actually later than your current x time. While this cannot happen with ordinary XEvents, delayed client messages may have this effect. Be prepared that evil implementations may even pass CurrentTime sometimes.

But I [OWT] wouldn't agree with this advice. The point of a timestamp is to make sure that when events are processed out of order, the event generated last by the user wins for shared resources such as input focus, selections, and grabs. An example of where this can matter is if you have

```
Toplevel Window
  Embedder
    Client
      Text Entry 1
  Embedder
    Client
      Text Entry 2
```

If the entries are set to select the text on focus in, and the user hits TAB in quick succession, then the timestamps on the FOCUS_IN events are what makes sure that Entry 2 actually ends up owning the PRIMARY selection, instead of it being a race between the two clients. But in situations like this having the correct timestamp only matters if a user action triggers the behavior.

Hence the advice that the timestamp should be the time from the event currently being processed.

If no explicit user action is involved, then the best thing to do is to use CurrentTime; using the timestamp from the last X event received can cause problems if the ultimate trigger of the behavior is a timeout or network and the last X event happened some time in the distant past.

Complexity of accelerator handling

The current specification for accelerator handling is a little complex. Most of the

complexity (the accelerator IDs) comes from the need to handle conflicting accelerators. GTK+ currently implements a simpler scheme where grabs are identified only by key symbol and modifier and conflicting mnemonic resolution doesn't work across embedder/client interfaces.

Infinite loops in focusing

There is the potential for infinite loops of focusing - Consider the case:

```
Toplevel Window
  Embedder
    Client
```

Where there are no focusable sites in the client or in the toplevel window. Then if Tab is pressed, the embedder will send: FOCUS_IN/FOCUS_FIRST to the client, the client will send FOCUS_NEXT to the embedder, the toplevel window will wrap the focus around and send FOCUS_IN/FOCUS_FIRST to the client...

The minimum mechanism that seems necessary to prevent this loop is a serial number in the FOCUS_IN/FOCUS_FIRST message that is repeated in a resulting FOCUS_NEXT message.

A possibly better way of handling this could be to make FOCUS_IN have an explicit response; that, is, add a XEMBED_FOCUS_IN_RESPONSE that the client must send to the embedder after receipt of a FOCUS_IN message.

Table 6. XEMBED_FOCUS_IN_RESPONSE

detail	1 if the client accepted the focus, 0 otherwise
data1	serial number from XEMBED_FOCUS_IN

The main problem with requiring a response here is that caller needs to wait for the return event, and to handle cases like parent (client 1) => child (client 2) => grandchild (client 1), it probably needs to process all sorts of incoming events at this point. If the user hits TabTab in quick succession things could get very complicated.

Robustness

The protocol, as currently constituted, is not robust against the embedder crashing. This will result in the embedder window being destroyed by the X server, and, as a consequence client's window being unexpectedly destroyed, which will likely cause the client to die with a BadWindow error.

To fix this requires an X protocol extension which extends the functionality of `XChangeSaveSet()` in two areas:

- Allow it to be specified that the saved window should be reparented to the root window rather than to the nearest parent. (The nearest parent typically being the window manager's frame window, reparenting to the nearest parent only saves the client until the window manager cleans up and destroys the frame window.)

- Allow it to be specified that the saved window should be unmapped rather than then mapped. (Without this capability the client will mapped as a child of the root window, which will be confusing to the user.)

Sensitivity

Toolkits such as Qt and GTK+ have a concept of disabled widgets. This notion is typically hierarchical, so if the embedder or a ancestry of the embedder becomes insensitive, widgets inside the client should be displayed as, and act insensitive as well.

Directional focusing

Some toolkits, such as GTK+, support, along with the standard concept of a focus chain, the idea of *directional focusing*; it's possible in some cases to navigate focus using the arrow keys. To do this perfectly, you need to have information about the coordinates of the original focus window, which is hard to do in an embedding context, but a good approximation is to, when focusing into a container, provide the side of the container where focus is coming from and to focus the "middle widget" on this side.

This could be supported by adding an extra data field to to the

XEMBED_FOCUS_FIRST/XEMBED_FOCUS_LAST subtypes of XEMBED_FOCUS_IN and to XEMBED_FOCUS_NEXT and XEMBED_FOCUS_PREV, which would contain:

```
/* Directions for focusing */
#define XEMBED_DIRECTION_DEFAULT    0
#define XEMBED_DIRECTION_UP_DOWN    1
#define XEMBED_DIRECTION_LEFT_RIGHT 2
```

Applications supporting only normal tab focusing would always pass XEMBED_DIRECTION_DEFAULT and treat all received directions as XEMBED_DIRECTION_DEFAULT.

The argument against supporting this is that it's a rather confusing feature to start with (many widgets eat arrow keys for other purposes), and becomes more confusing if you have a application containing widgets from different toolkits, some of which support it, some of which don't.

Modal dialogs

The specification doesn't have any provisions for handling the case where an embedded client wants to put up a dialog. Such a dialog should be transient-for the real toplevel window, and, if modal, should block the entire toplevel window. To fully implement this, you would need some concept of an application that spanned multiple toplevel windows in multiple clients.

Propagation of key presses

It's frequently useful to have key bindings that trigger on a widget if the focus is on a child of that widget. For instance, ControlPageUp and ControlPageUp switch pages in a notebook widget when the focus is on a child of the notebook. The XEmbed spec currently has no handling of this situation.

The simplest solution would be to specify that if the client widget doesn't handle a key press sent to it, it then sends the event back to the embedder. Some care would be required in the embedder handle infinite loops, but it shouldn't be that bad.

Handling of toplevel modes

GTK+-2.0 contains a feature for key navigation of tooltips where Control-F1 toggles a "tooltips keyboard mode" where the tooltip for the currently focused window is displayed. There is no way of propagating this across XEMBED. This feature could clearly be implemented the same way as XEMBED_WINDOW_ACTIVATE, but adding a pair of messages for every feature of this type seems excessive.

A possible alternate idea would be to add a _XEMBED_STATE property that the embedder sets on the client window which is a list of atoms. This could actually be used to replace XEMBED_WINDOW_ACTIVATE, and XEMBED_MODALITY_ON, simplifying the protocol.

There are some race conditions in maintaining this property if the client is allowed to reparent itself out of the embedder that would have to be considered.

A. Change history

"Version 1.0 DRAFT 1", 22 April 2000, Matthias Ettrich.

"Version 1.0 DRAFT 2", 15 August 2000, Matthias Ettrich.

Version 0.5, 19 April 2002, Owen Taylor.

- Add the life-cycle chapter, including _XEMBED_INFO property, and the XEMBED_MAPPED flags.
- Define the data2 for XEMBED_EMBEDDED_NOTIFY to be the protocol version in use.
- Replaced XEMBED_GRAB_KEY scheme for handling accelerators with XEMBED_REGISTER_ACCELERATOR.
- Removed text "This also means that the client has to prepare for becoming visible anytime without filing a map request itself" from the description of XEMBED_EMBEDDED_NOTIFY".

- Added text about the independence of FOCUS_IN/OUT and ACTIVATE/DEACTIVATE to the description of XEMBED_WINDOW_ACTIVATE / XEMBED_WINDOW_DEACTIVATE.

- Added note about fields in WMNormalHints not necessarily being obeyed by embedders.

- Removed mention of XEMBED_PROCESS_NEXT_EVENT, which is no longer part of the protocol.

- Added definitions of "Active" and "Focused" to the definitions section.

- Added issues and change history sections.

- Lots of textual editing for clarity, style consistency.

- Converted to docbook format.