Why X Is Not Our Ideal Window System

Hania Gajewska
Mark S. Manasse
DEC Systems Research Center
130 Lytton Ave. Palo Alto, CA 94301

Joel McCormack
DEC Western Research Laboratory
100 Hamilton Ave. Palo Alto, CA 94301

Abstract

Extensive experience with X11 has convinced us that it represents a true advance in window systems, but that there are areas in which the X protocol is seriously deficient. The problems we describe fall into seven categories: coordinate system pitfalls, unavoidable race conditions, incomplete support for window managers, insufficient window viewability information, difficulties with interactive mouse-tracking, pop-up and redisplay inefficiencies, and exceptional condition handling. We propose solutions for most of these problems. Some solutions could be easily incorporated into the X11 protocol. Other proposals are too incompatible to be adopted, but are nonetheless included for the benefit of future window system designers.

---

1. Introduction

We come to praise X, not to bury it. Our combined experience with X11 encompasses the design and implementation of several window managers, the design and implementation of the Xtk toolkit intrinsics, ports of the X server, and protocol converters between X and other windowing systems.

This experience has convinced us that X represents an advance in window systems: it is network-transparent, it runs on a wide variety of graphics hardware, and it provides an efficient connection to the capabilities of the underlying hardware. The same experience has also convinced us that X is not a perfect window system. In this paper we limit ourselves to specific problems with the X11 protocol [7]; we do not complain about the philosophy or imaging model underlying X, nor do we bemoan the fact that X is not NeWS [3], NeXTstep [9], PostScript [1], or PHIGS [6].

The problems we have encountered fall into seven categories:

1. The mixing of signed and unsigned coordinates causes problems both in the protocol, where 3/4 of the coordinate space is often unrepresentable, and in the C language bindings.
2. The X protocol is asynchronous for efficiency: in general, neither the server nor clients wait for replies. But the protocol's synchronization mechanisms are insufficient, and leave many unavoidable race conditions.
3. The X protocol attempts to be policy-free and tries not to dictate any particular style of window management. However, some desirable window manager features cannot be implemented correctly, because there are window attributes which the window manager can neither fetch nor monitor.

4. The X protocol provides visibility notification events so that clients can avoid computation of obscured window contents. However, this notification doesn't work well for nested windows or for windows with backing store.
5. None of the several ways that an application can implement interactive mouse tracking of crosshairs, bounding boxes, etc., allow both efficiency and correctness.
6. Popping up menus and dialog boxes is slow because it requires too many round trips and generates too many events. Repainting when portions of a window become visible is often slow.
7. Exceptional conditions are poorly handled. Faulty programs can freeze the server, and clients cannot kill queued requests if the user doesn't want to wait for the server to finish servicing them.

Although we define some X-specific vocabulary, we recommend keeping handy a glossary of X terms, such as the one in Reference [7], while reading this paper.

2. Coordinate Representation

There are three problems in X11's definition of coordinates: positions and sizes have different physical representations, window borders introduce inconsistent views of a window's coordinate system, and the restriction that window sizes must be positive leads to unnecessary special cases.

2.1. Signed positions vs. unsigned dimensions

The X11 protocol defines x and y coordinates as 16-bit *signed* integers, and width and height dimensions as 16-bit *unsigned* integers. A window, or any other rectangle, is defined by the signed position of its north-west corner and the unsigned dimensions of its width and height. This makes sense intuitively: the coordinates of the north-west corner of a window may be beyond the boundaries of its parent window and thus negative, while width and height are generally thought of as nonnegative values.

However, using all 16 bits to create a very tall window is pointless, because many positions in the window cannot be addressed using signed 16-bit values. For example:

- All text strings must begin in the top half of the window.
- All rectangles and subwindows must have their upper left corner in the top half of the window.
- All lines and polygons must lie entirely within the top half of the window.

Similar restrictions apply to output to very wide windows (figure 2-1).

There are even graver problems with input. Any event that uses the signed 16-bit representation for positions will be reported correctly only if it occurs in the top half of a tall window. This includes `ButtonPress`, `ButtonRelease`, `MotionNotify`, `KeyPress`, `KeyRelease`, `EnterNotify`, and `LeaveNotify`. The events `Expose` and `GraphicsExpose` are an exception: since the exposed rectangle is always contained within the window, the x and y fields are always positive, and the protocol defines these fields to be 16-bit *unsigned* values.

If an X server allows a client to create very large windows, the server may be forced into a

situation in which it either sends an event that contains incorrect information, or doesn't deliver the event at all. Thus we believe that the protocol implicitly disallows very large windows; it allows only off-screen pixmaps, which do not have input events, to be large.

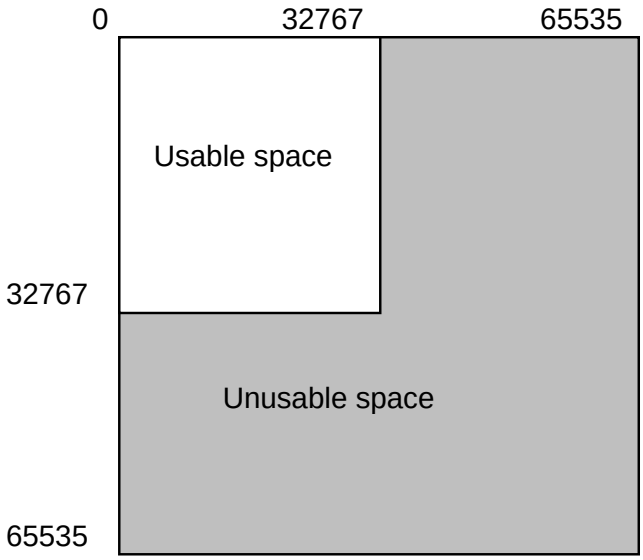

Figure 2-1: The X coordinate system: definition vs. reality

The `TranslateCoordinates` request causes even more problems. `TranslateCoordinates` converts signed x and y values relative to a source window into signed x and y values relative to a destination window. It is easy to construct cases in which the returned x and y values are incorrect—unless the source and destination window have the same origin relative to the root window, there is always a range of values which cannot be translated correctly.

Even if we assume that the given x and y values are contained within the source window, or that the returned x and y values are contained within the destination window, we can still construct problematical cases. In order to guarantee that returned values fit into a 16-bit signed number under these assumptions, all windows must be entirely contained within the signed 16-bit coordinate space of the root window.

Though we are critiquing the X protocol itself, and not particular language bindings, we next show that the choice of `unsigned int` to represent dimensions in the Xtk toolkit [4] and Xlib [7] interacts poorly with C language semantics.

Consider a window manager that wants to constrain windows to have at least one pixel on the screen. Checking that the east edge of the window isn't to the west of the screen seems easy:

```
w.x + w.width > 0
```

This doesn't work for two reasons; here we consider C's arithmetic semantics, and defer the second problem to the discussion of window borders. C defines arithmetic operations that include an unsigned operand to have an unsigned result. Since a window's width is an unsigned value, the test above succeeds unless the sum is exactly 0. This type of mistake has surfaced repeatedly, both in applications and in the sample server implementation.

Recommendation: Width and height should be defined as either 15-bit unsigned values, or better, as 16-bit signed integers with negative values treated as errors. In the C Xlib and Xtk bindings, width and height should be signed to match C arithmetic semantics. In languages with subrange types, their type should be defined as the range 0..32767. The protocol should also explicitly restrict all windows to be contained within the 16-bit signed coordinate space of the root window.

2.2. Window coordinates and borders

Rewriting the example above as

```
(int)(w.x + w.width) > 0
```

is still incorrect. The protocol's definition of a window's position is inconsistent with the window's dimensions. The x and y coordinates of a window specify the position of the outer north-west corner in the window's parent, while the height and width specify the window's inner dimension. The space between is occupied by the window's border.

Thus, the correct way to assure that some part of the interior of a window w isn't to the west of the screen:

```
(int)(w.x + w.border_width + w.width) > 0
```

Defining x and y as the coordinates of the outer corner has led to many bugs in server implementations and in clients, because it is inconsistent with width and height being inner dimensions, and because it means that the coordinates (0,0) inside a window describe a point different from the coordinates of the north-west corner of the window in the window's parent. On the other hand, we suspect that defining x and y as the inner corner would lead to an equivalent number of bugs in other places.

Confusion seems unavoidable, since the protocol describes two views of the window with one set of coordinates. The external view of the window includes the border: its natural origin is the outer north-west corner, and its natural size is the inner width and height plus twice the border width. The internal view of the window omits the border: its natural origin is the inner north-west corner, and its natural size is the inner width and height. These two views are compressed into the five numbers x, y, width, height, and border width. No matter how these numbers are assigned, one or both views of the window will be awkward to deal with.

The real problem lies at a more fundamental level. The holdover of borders from the X10 protocol is a mistake. Borders cause complications in the protocol, servers, and clients, but don't provide enough payback to justify their existence. Many windows don't use borders at all. Windows that use borders to highlight could paint bordering rectangles explicitly. Applications that carefully align the position of subwindows to use their borders as separators could just as easily leave small gaps between the windows to let a background color shine through. Those few windows that need borders can be embedded inside container windows. This would be more efficient if the window gravity of the embedded window could be set so that the simulated border size is preserved when the parent is resized.

Recommendation: We can't justify removing borders from the X protocol, given the large number of clients that would need modification. However, future window systems should let clients deal with window borders. Programmers using X11 should take care when performing

geometry computations.

2.3. Window sizes must be positive

Although 0 is a natural lower bound for dimensions, it is not the one that the X11 protocol adopts: specifying 0 for the width or height of a window generates an error. Because parent windows must be created before child windows, 0 is the natural dimension value when a parent's size depends on the number and sizes of its children. Instead, some meaningless numbers must be supplied for width and height. Further, the parent's size cannot just drop to 0 when its children go away; instead, an application must make removing the last child a special case by unmapping the parent and remembering that the parent's current size does not reflect the contents.

Applications try to create windows with a 0 width or height so often that the Xtk toolkit specifically checks for this case before creating a window, in order to return an understandable error message.

Recommendation: Permitted widths and heights of windows should include 0.

3. Race Conditions

Non-distributed window systems use simple synchronization mechanisms between user, graphics display, and application. For example, a Macintosh application polls for user events, and paints using synchronous procedure calls [2]. This simple style makes it natural for applications to delay polling for events until the screen is in a known, consistent state.

The X protocol must work well even when the server and its clients are running on different machines. Round-trip times to send a message to the server and get a reply may take hundreds of milliseconds, so synchronous painting and event polling are infeasible. X11 therefore defines an asynchronous protocol: a client sends a stream of painting and window management requests to the server, while the server sends a stream of events back to the client. Ordinarily, the two streams are not synchronized with each other. Although the server processes requests from a single client in order, in general no order is defined for interleaving requests from different clients.[1]

The protocol defines no particular window management scheme, but provides hooks so that a window manager, which is simply an X11 client program, can monitor and benignly interfere with requests issued by more ordinary applications. Such applications must contend with the possibility that a window manager may delay or modify some requests, while other requests will execute immediately.

Several types of race conditions arise from the asynchronous request and event streams, and from the window manager hooks. Sometimes things happen in the expected order, sometimes they don't.

In the following sections, we expose several classes of race conditions. We show how some race conditions can be avoided using synchronous grabs, timestamps, or event notification, and

[1]Actually, the protocol never explicitly states that the server processes requests in the order they are sent, but we have assurances from the designers that this is certainly intended.

suggest how these mechanisms can be extended to eliminate the rest of the races we describe.

Although some of the races can be avoided within the framework of the X11 protocol, doing so is hard. Naive programmers may be unaware of the races inherent in seemingly straight-forward programs until some change in the environment triggers a race. Our tripping over such races led to many of the conventions comprising the Inter-Client Communication Conventions Manual [7], commonly referred to as the ICCCM.

We make extensive use of modified Feynmann diagrams to show independent actors (server, clients, window manager, and user) and their interactions through time. Contrary to physicists' conventions, time flows down the page. To keep things simple, the user and server timelines are combined—we assume that the server queues raw mouse and keyboard actions immediately.

3.1. Races within a client

The asynchronous request and event streams cause the simplest race conditions. Consider an application that pops up a menu when the left mouse button is pressed, and performs the action pointed to by the mouse cursor when the button is released. The following actions will take place:

1. The user presses the mouse button.
2. The server sends a `ButtonPress` event to the application.
3. The application issues a `MapWindow` request to map the menu.
4. The server maps the menu.
5. The server sends `Expose` events to the application.
6. The application paints the contents of the menu.

Figure 3-1 shows what happens if the user releases the button before step 4. Since the menu window has not yet appeared, the server sends a `ButtonRelease` event that contains whatever window is below the mouse at the time. This *mouse-ahead* race can occur if the user is experienced and knows how far to move the mouse to select a common operation, but the application is running on a busy machine or over a slow network link.

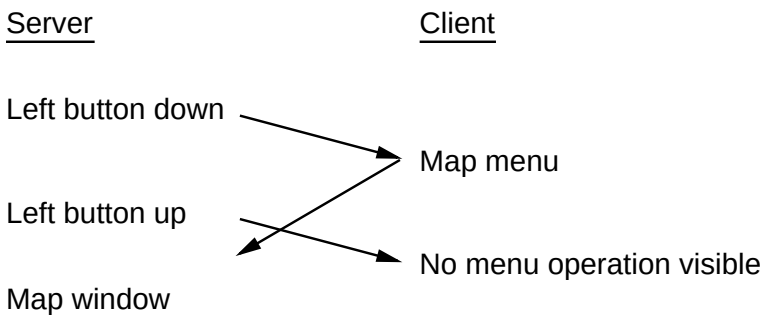

Figure 3-1: Pop-up menu race

Because pressing a key or a mouse button may initiate work that should be completed before any more user events are interpreted, the X11 protocol contains the concept of a *synchronous grab*. When a synchronous grab activates, the server stops sending mouse and keyboard events

to the client. The server still queues the events internally, but keeps them in a raw, uninterpreted form. In particular, the server does not compute the window in which the event took place until the client allows it to do so by calling `AllowEvents`.

A client can set a grab to activate on any window, button or key, and modifier combination. In this example, the client sets a grab to activate when the left mouse button is pressed in the window that pops up the menu. To ensure that mouse events are processed in the proper context, the client first maps the menu, then issues an `AllowEvents` request. When the server receives the `AllowEvents` request, it interprets queued events using the current arrangement of windows, and then dispatches the events. Figure 3-2 shows how a synchronous grab solves the mouse-ahead problem.

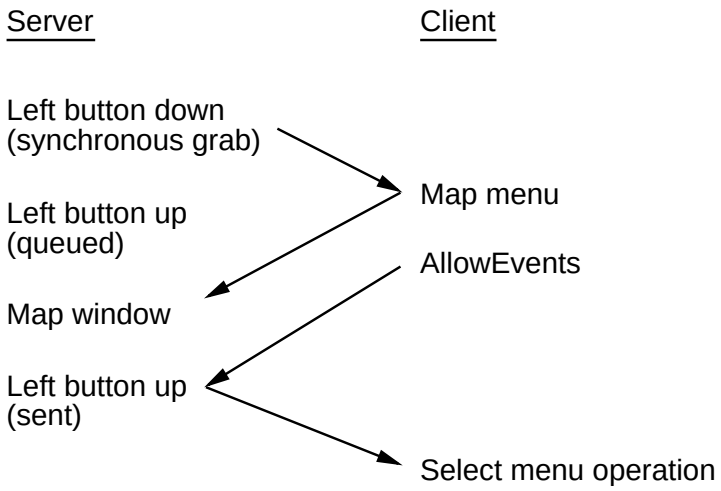

Figure 3-2: A synchronous grab solves the pop-up menu race

Note that a similar race condition exists for popping up a dialog box and setting keyboard input focus to the box. In this case the problem is type-ahead rather than mouse-ahead. The client and server actions are:

1. The user presses the mouse button.
2. The server sends a `ButtonPress` event to the application.
3. The application uses `MapWindow` to map the dialog box.
4. The application uses `SetInputFocus` to set keyboard focus to the dialog box.
5. The server maps the dialog box.
6. The server sends `Expose` events to the application.
7. The server sets input focus to the dialog box.
8. The application paints the contents of the dialog box.

If the user types any characters before step 7, the characters are dispatched to the wrong window. The solution is familiar—use a synchronous grab—but subtle complications arise. We discuss window manager race conditions in more detail below, so here we just note that menus are usually override-redirect windows, and are thus immune to window manager interactions; dialog boxes are often normal windows, so the window manager may delay execution of the `MapWindow` call. Because the protocol does not allow input focus to be set to an unmapped

window, the application must first wait for the server to send a MapNotify event. Only then can it call SetInputFocus and AllowEvents. The complete solution is shown in figure 3-3. (This solution may actually be incorrect under some window managers; other possible solutions are described in a later section.)

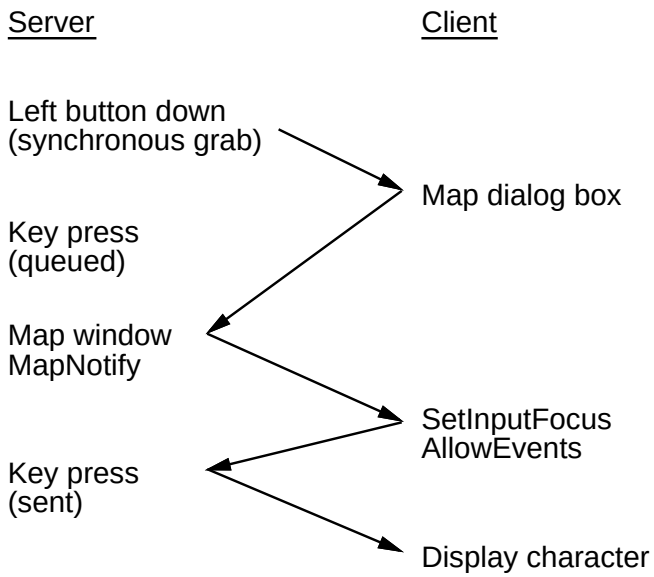

Figure 3-3: A synchronous grab solves the pop-up dialog box race

A similar intraclient race condition cannot be solved with the existing synchronous grabs. Suppose we want to implement a Macintosh-style menu bar. Such a menu bar puts up a menu when a mouse button is pressed in any of its subwindows. While the mouse button remains down, the old menu goes away and a new menu appears whenever the mouse enters a different subwindow of the menu bar.

Implementing menu bars in X is almost like implementing pop-up menus, and is subject to a similar race condition if the user releases the mouse button before the server maps the menu. An EnterNotify event tells the client when the mouse moves into a different subwindow of the menu bar; this serves the same role as ButtonPress in a pop-up menu.

Setting a synchronous grab on EnterNotify events would solve the race condition, but grabs are permitted only on key and mouse button presses. The obvious change to the protocol is to allow EnterNotify events to activate synchronous grabs. In other situations, race conditions arise because grabs aren't allowed on LeaveNotify events, so the protocol should allow these events to activate synchronous grabs as well.

Since EnterNotify and LeaveNotify events differ from ButtonPress events—for example, multiple EnterNotify events may be generated for a single user action—event freezing on EnterNotify and LeaveNotify must be slightly different from grabs on ButtonPress. We'll still refer to this event freezing as a synchronous grab in order to emphasize the similarities.

While synchronous grabs are necessary to avoid race conditions, they may introduce performance problems. Many implementations of menu bars avoid menu flashing to improve

responsiveness: if an `EnterNotify` event is immediately followed by a `LeaveNotify` event, these implementations assume that the user is sweeping the mouse across the menu bar, and discard both events. If each `EnterNotify` event causes a synchronous grab, then a `LeaveNotify` event cannot occur until the pop-up window is mapped and the client calls `AllowEvents`. The resulting performance degradation may outweigh the benefits of correctly handling mouse-ahead.

We know of no solution to this problem within the spirit of the protocol. Later in this paper we provide arguments for `EnterNotify` and `LeaveNotify` grabs that are untainted by performance questions.

3.2. Races between two clients

More complex race conditions occur when two clients conflict over a single resource, such as keyboard focus or the current selection. The simplest race involves unwarranted assumptions about ownership of a unique resource. The server notifies clients when they have lost ownership of input focus and other unique resources, but absence of such notification should be used as a hint, not as the truth.

Consider two applications that assign keyboard focus to their respective windows whenever the left mouse button is pressed. One application runs on a fast machine and currently owns input focus. The other application runs on a slow machine.

The user clicks in the slow application window to give it focus, but before the slow application transfers focus, the user clicks back in the original (and still current) focus owner, and continues typing there. The fast application processes its click first, but since it erroneously believes that it already owns the input focus it doesn't bother to issue another `SetInputFocus` call. Eventually the slow application processes its click and calls `SetInputFocus`, taking focus away from the fast application and subverting the user's intent. Figure 3-4 shows this race condition.

Thus, applications should never optimize away requests for a unique resource just because they already own it. But how do they properly retain ownership? If we change the ''Ignore event'' to ''`SetInputFocus`'' in figure 3-4 there are still orderings that ultimately assign input focus to the slow application.

The X11 protocol uses *timestamps* to deal with inter-client race conditions. Several events contain a timestamp, as do requests that establish ownership of a unique resource. For the remainder of this section we use `SetInputFocus` as a specific example of this class of requests.

A timestamp is ideally a monotonically increasing function. In the case of an X server, a timestamp is a 32-bit value containing the number of milliseconds elapsed since the server was booted.[2] When the user presses a key or mouse button, the current server time is attached to the raw event.

---

[2]Timestamps wrap around about every 50 days, but are interpreted correctly as long as the time between two events doesn't exceed 25 days.

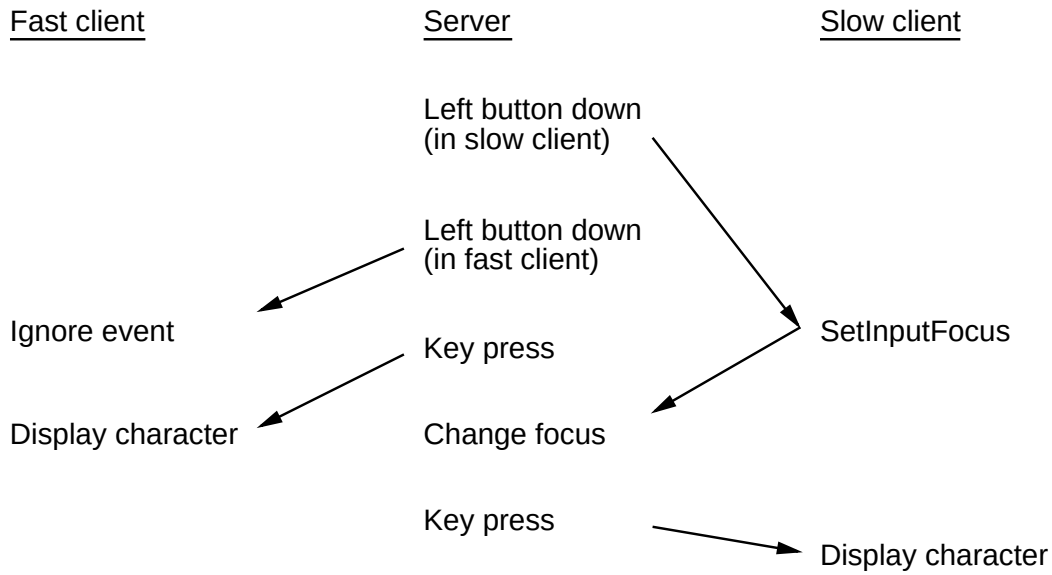

Figure 3-4: Input focus race between clients

When an application issues a `SetInputFocus` request in response to an event, it should also supply the timestamp from the event. The server compares the timestamp in the `SetInputFocus` request with the *last-focus-change time*, which is the timestamp from the last successful `SetInputFocus` call. If the new timestamp is earlier than last-focus-change time, the server ignores the request—the current call has already been superseded by a `SetInputFocus` call issued in response to a later event. If the new timestamp is later or the same, the server changes input focus and updates the last-focus-change time.[3] Timestamps thus ensure that the ordering of user events is maintained in the corresponding allocation of unique resources. Figure 3-5 shows how timestamps solve the input focus race between clients.

There is another race condition lurking in this example. If the user clicks in the slow application, types a few characters, and then clicks back in the fast application, the server sends all characters to the fast application. We already know how to solve this problem: set a synchronous grab on the button press, and call `AllowEvents` after calling `SetInputFocus`. Figure 3-6 shows the ultimate solution.

Using a synchronous grab exacts a performance penalty—the fast application can't display characters meant for it until the slow application calls `AllowEvents`. But the grab does guarantee that each keystroke goes to the correct window. The X protocol does not dictate what philosophy, if any, an application writer should adopt toward race conditions; the safest method is often not the most efficient. Caveat implementor.

`InstallColormap` and `UninstallColormap` are susceptible to the same type of race conditions as `SetInputFocus`, but these race conditions are unavoidable, because colormap requests do not take timestamps.

[3] In a later section, we exploit the fact that `SetInputFocus` takes effect if the supplied timestamp is equal to last-focus-change time.

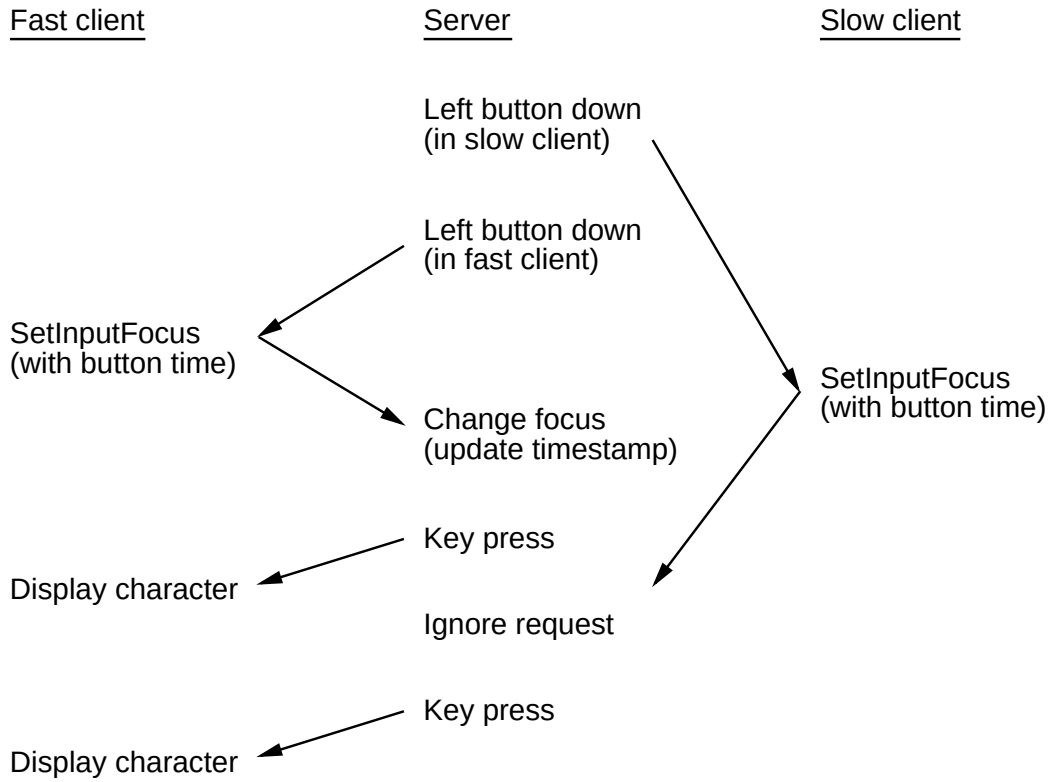

Figure 3-5: Timestamps solve the input focus race

The ICCCM attempts to resolve colormap installation races by legislating conventions that clients must adhere to. A better solution is to add the appropriate timestamps, so that application writers can use similar mechanisms to solve similar problems. The interpretations of these timestamps must differ slightly from that in `SetInputFocus`, because some servers allow multiple colormaps to be installed simultaneously.

Recommendation: `InstallColormap` and `UninstallColormap` should take a timestamp parameter. `InstallColormap` would use the timestamp to ensure that the most recent colormaps are physically installed. `UninstallColormap` would use it to ensure that it doesn't erroneously uninstall a colormap after another client has installed it with a more recent timestamp.

3.3. Client-side races with the window manager

Much unexpected behavior results when clients perform operations that work with no window manager or a simple window manager, but do not work with a *reparenting* window manager. This section describes race conditions nominally caused by a client program doing the wrong thing. Since the client actions causing these race conditions are so natural, we point out protocol changes that would make the window manager responsible for, and capable of, eliminating these races.

X11 allows a privileged client, the window manager, to oversee and manage the behavior of top-level client windows. Simple X11 window managers ask for notification events in order to monitor client behavior, then react to these actions after the fact. For example, when the server

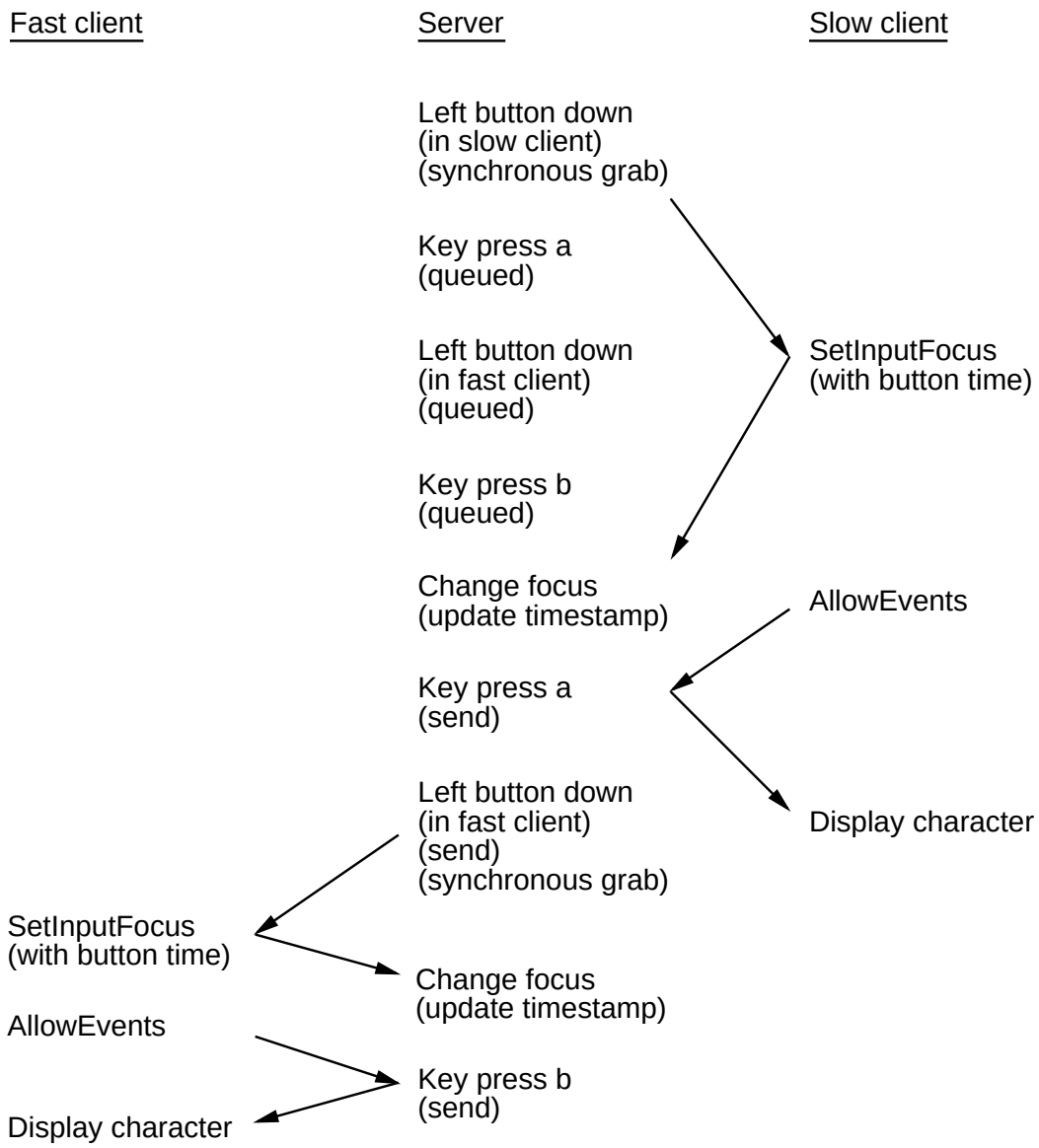

Figure 3-6: The ultimate input focus race solution

notifies the *uwm* window manager that a client has mapped its top-level window, *uwm* unmaps a corresponding icon window.

Most window managers, and particularly reparenting window managers, play a more active role. These *redirecting* window managers instruct the server to redirect all top-level `MapWindow`, `ConfigureWindow` and `CirculateWindow` requests made by other clients. The server doesn't execute such a request, but sends a special redirection event to the window manager. Since the server maintains no information about a request after redirecting it, the window manager must assume complete responsibility for the request. It may ignore the request, grant the request by issuing an identical request itself, or translate the request into one or more other requests.

For example, the *twm* window manager decorates windows with a title bar. To do so, *twm* creates a frame window big enough to hold both the client window and the title bar. It then

reparents the client window to be a child of the frame. *twm* uses redirection so that it can translate `MapWindow` and `ConfigureWindow` requests on the client window into `MapWindow` and `ConfigureWindow` requests on the frame window. If instead *twm* waited for notification events, `ConfigureWindow` requests in particular would cause ugly screen flashing before the window manager got things straightened out.

Redirection causes problems for naive clients. Imagine an application that starts painting as soon as it maps a window. If `MapWindow` is not redirected, all is well. But under a redirecting window manager, the server sends the map request to the window manager, the client starts painting before the window manager processes the request, and the server ignores all painting calls until the window manager finally maps the frame window. Figure 3-7 illustrates this scenario.

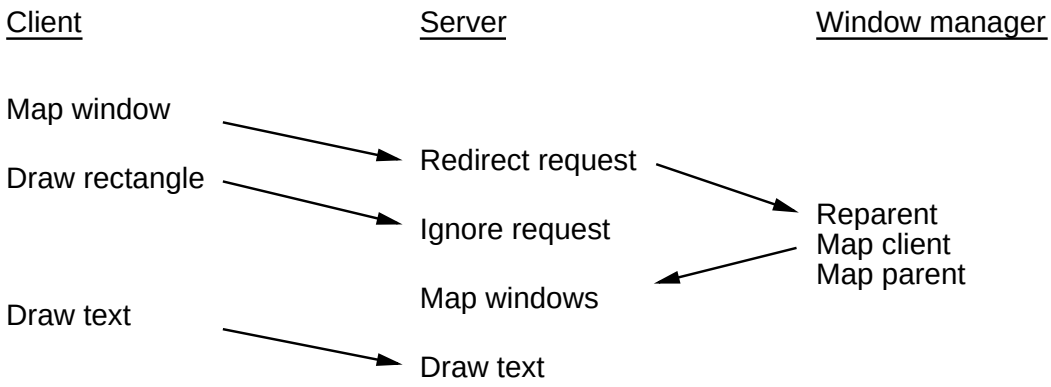

Figure 3-7: Map window race

Clients can avoid this problem if they wait for a `MapNotify` event (or for `Expose` events) before they begin painting, as shown in figure 3-8.

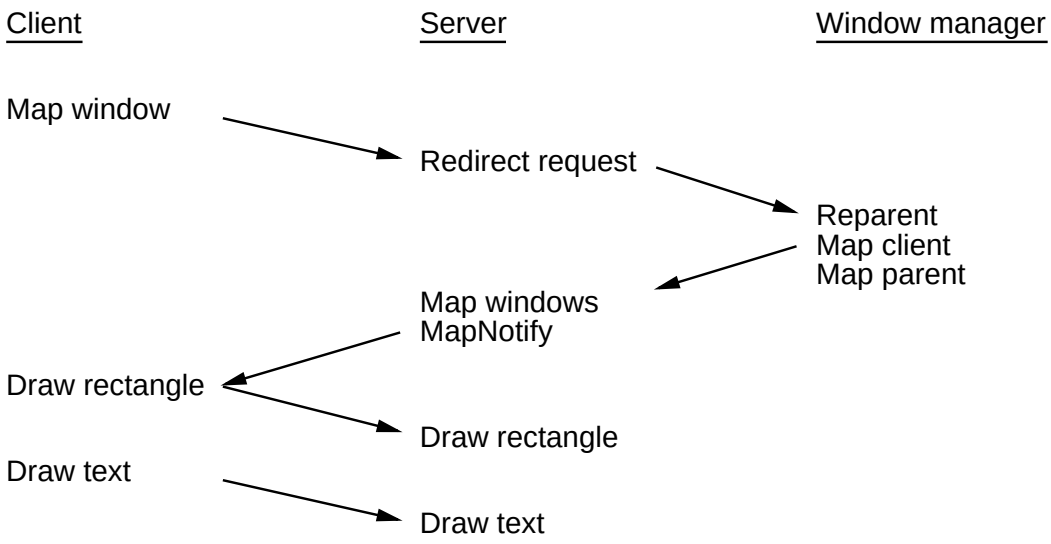

Figure 3-8: MapNotify solves the map window race

The solution shown in figure 3-8 may not actually solve the problem, depending on the window

manager involved. In fact, we have deliberately depicted the behavior of most window managers, which introduces a low-probability race condition. These window managers map the client window before mapping the frame window in order to improve efficiency and reduce screen flashing. Though unlikely, the following scenario may occur:

1. The window manager sends a `MapWindow` request for the client window.
2. The window manager sends a `MapWindow` request for the frame window.
3. The server maps the client window and sends a `MapNotify` event to the client, but doesn't yet map the frame window.
4. The client receives the `MapNotify` event for the client, and sends painting requests in return.
5. The server ignores several painting requests, because the window is not *viewable*. It is not enough for a window to be mapped; to be viewable all of its ancestors must be mapped as well.
6. The server finally maps the frame window.

Waiting for `MapNotify` is a fairly common X11 idiom, particularly for clients that set input focus to dialog boxes. Thus, we recommend that either the protocol or the ICCCM instruct window managers to avoid this scenario using one of two techniques. By using a background of `None` in the frame window to avoid screen flash, a window manager can map the frame before the client window; this is less efficient than the reverse order, but probably insignificantly so. Alternatively, a window manager can grab the server, issue the two `MapWindow` requests, and ungrab the server, which ensures that the server maps both windows before processing any other requests.

If the protocol and ICCCM remain silent on this issue, clients must compensate by waiting for some event other than `MapNotify`. In the example above, the client should wait for `Expose`.

Several other requests require that the specified window be mapped or viewable. The client must wait for an appropriate event before issuing such requests. `SetInputFocus` requires the window to be viewable; if a client pops up a dialog box and calls `SetInputFocus` before receiving `VisibilityNotify`, the dialog box may not get focus. Similarly, `GrabPointer` fails if either of its window parameters are not viewable. `UnmapWindow` requires that the window already be mapped; if a client calls `UnmapWindow` before receiving `MapNotify`, the window may remain mapped.

`SetInputFocus` is especially pernicious—if the window is unviewable, an X error is generated. Since few clients contain useful error handlers, most will terminate after printing the X error. Yet the condition that generates this error is likely to occur whenever there is a race between the application setting the focus and the user iconifying the window—not an unlikely occurrence. In general, it seems inadvisable to use errors to report failures due to unavoidable races.

In order to operate correctly under a redirecting window manager, almost all clients must ask for and wait for `Expose`, `MapNotify`, and `VisibilityNotify` events on their top-level windows. The ineffectiveness of this solution is evident from the volume of bug reports related to this issue that appear on the X mailing lists and news groups, as well as the volume of questions from our co-workers. The real problem is that application writers expect the ordering of their

request stream to reach the server unaltered, but redirection violates this assumption.

These problems could be solved better by making redirection synchronous: when the server redirects a client's request, it suspends processing of further requests from that client. Synchronous redirection requires adding a client identifier to all redirected requests, and adding an `AllowRequests` request. When the window manager has serviced the synchronously redirected request, it permits the server to continue processing client requests by issuing an `AllowRequests` request, supplying the client identifier contained in the redirected request.[4] Figure 3-9 shows how this scheme allows the natural expression of client programs.

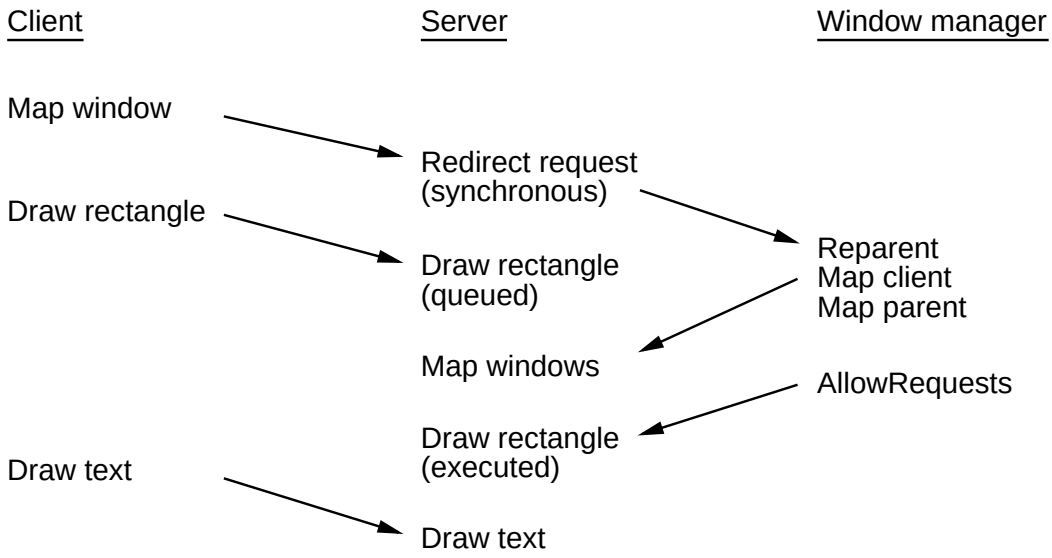

Figure 3-9: Synchronous redirection solves the map race

Recommendation: All redirection in X should be synchronous. Synchronous redirection in the example above greatly enhances the clarity and simplicity of client programs. For this reason alone, we would consider it justified. But synchronous redirection is good for much more than that. Below, we point out several more requests that should be redirectable. If the current protocol definition of redirection is used, making these requests redirectable solves a limited (but important) set of problems. If these requests are made synchronously redirectable, they solve a much larger set of problems. And, as explained in the section on efficiency, several time-critical operations can be made faster because synchronous redirection doesn't reorder request execution. Finally, `SetInputFocus` should fail silently if the window is unviewable, as it does if the timestamp is out-of-date.

3.4. Window manager races solved with redirection

The window manager is responsible for more than shuffling windows around: it must be able to impose a consistent style on clients that have differing ideas of how unique or limited resources should be managed. Thus the window manager may also be an input focus manager and a

[4]Efficiency considerations discussed later in this paper force a slight modification to the conditions under which the server suspends processing of events, but this doesn't affect the basic semantics.

colormap manager. However, several unavoidable race conditions prevent a window manager from imposing certain styles of management. These races are not the fault of the window manager or the client, but are inherent in the protocol.

There are two styles of keyboard input focus in X11: focus can be explicitly attached to a window, or it can automatically follow the mouse pointer. Attaching focus to a window is often called *click-to-type* because users generally set focus by pointing at a window and clicking a mouse button. Letting focus follow the mouse pointer is referred to as `PointerRoot` in the protocol. Debates about the virtues of these two styles are as vehement and pointless as debates about byte order.

A user may be running a mix of applications, where some explicitly set focus and others never do. Applications that never set focus themselves depend on the window manager (or the default `PointerRoot` model) for focus control. Applications that set focus explicitly may allow keyboard input to change the focus, as when the tab key moves the focus from field to field of a form window. These applications may also contain windows that don't ever get focus, such as scroll bars and push buttons.

Neither clients nor window managers have sole responsibility for setting input focus. Instead, setting the focus is a distributed responsibility. Clients can use `SetInputFocus` to indicate their focus desires. A simple window manager allows clients to fight it out, with generally poor results. A more complex window manager monitors `FocusIn` and `FocusOut` events, and may generate `SetInputFocus` calls itself, in order to provide the user's preferred focus model.

But, as shown in figure 3-10, a window manager cannot perfectly impose `PointerRoot` semantics over clients that set focus explicitly. Imagine a user running a window manager that tries to maintain `PointerRoot` focus at all times. The user clicks in a window to set an insertion point in a block of text. The application uses click-to-type, so the click also sets the input focus to the window. The user quickly moves the mouse to another window and types some characters, expecting the characters to go to the new window. But the window manager finds out about the input focus change after the fact, and only then can it force the focus back to `PointerRoot`. If the user is fast enough, the server sends the first few keystrokes to the window that set focus, rather than to the window containing the mouse pointer.

This race can be solved by making `SetInputFocus` a redirectable request, so that the window manager gets control before focus is actually changed. This is shown in figure 3-11.

The astute reader may notice that this solution has the potential for reintroducing a race condition that we have previously solved. The strict `PointerRoot` window manager of figure 3-11 ignores all focus requests, which is okay. But a hybrid window manager might permit an application to explicitly set focus as long as the mouse cursor is within the application's top-level window or any of its subwindows. This hybrid window manager resets focus to `PointerRoot` only when the cursor exits the top-level window (or alternatively, enters a new top-level window).

Unfortunately the *only* safe thing a window manager can do with a redirected `SetInputFocus` call is to ignore it. If the window manager grants a client's `SetInputFocus` request and issues a `SetInputFocus` call on the requested window, then an application cannot correctly handle type-ahead. We previously solved this type-ahead race by using synchronous grabs and `AllowEvents`. If `SetInputFocus` can be redirected, the

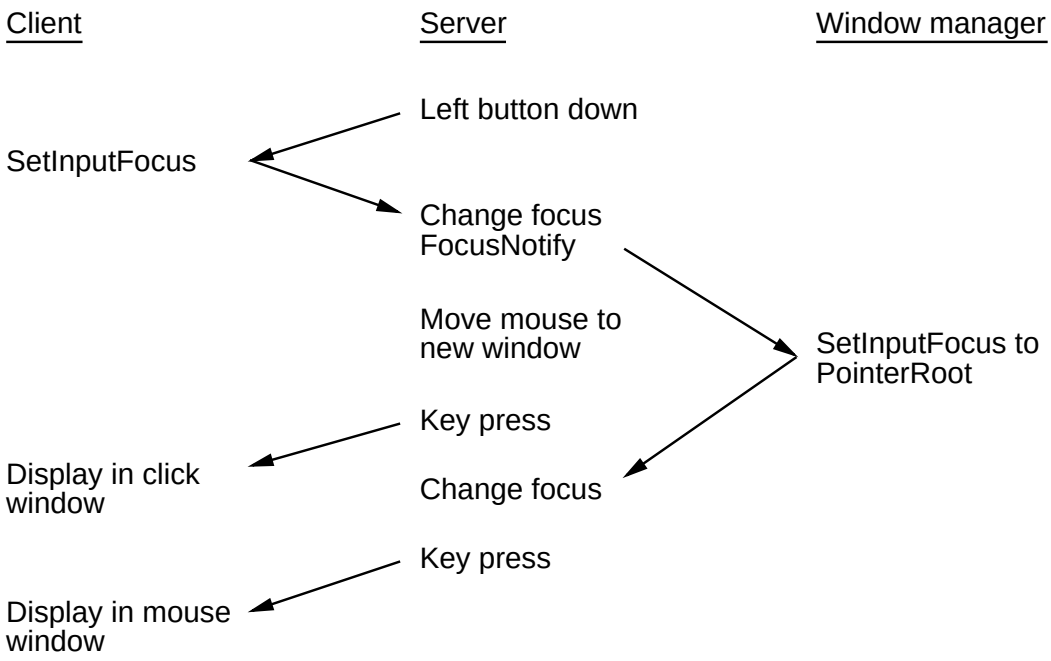

Figure 3-10: PointerRoot focus race

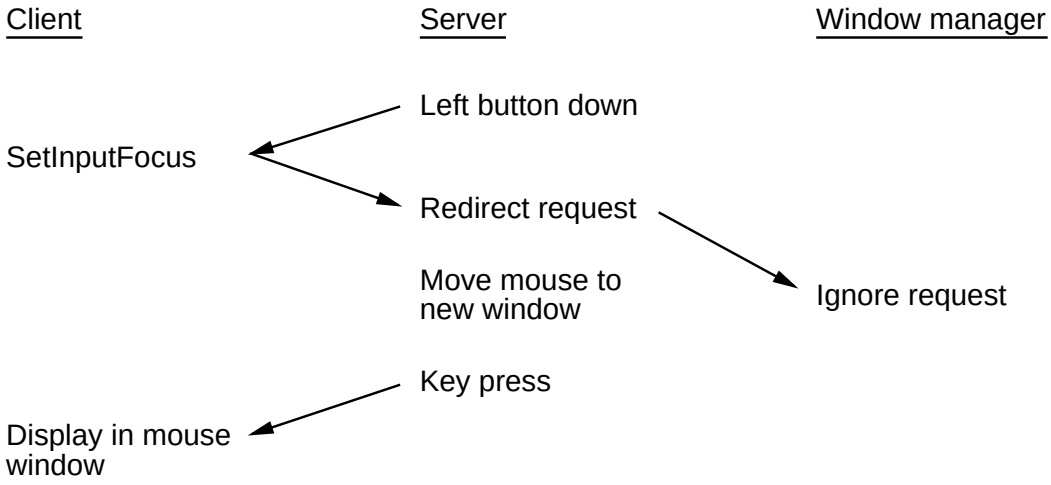

Figure 3-11: Redirection solves the PointerRoot focus race

client should wait for `FocusIn` before calling `AllowEvents`. But the client cannot wait for `FocusIn`, because it won't receive such notification if the window manager doesn't permit focus to change, or if the window already owns the input focus.

If `SetInputFocus` requests were synchronously redirected, this race condition could be avoided (figure 3-12). Synchronous redirection guarantees that a client's requests are not reordered. This allows the client to call `SetInputFocus` and `AllowEvents` in succession. By the time the server executes the `AllowEvents` request, the window manager has dealt with the input focus.

Finally, a hybrid window manager must be able to properly restore `PointerRoot` focus

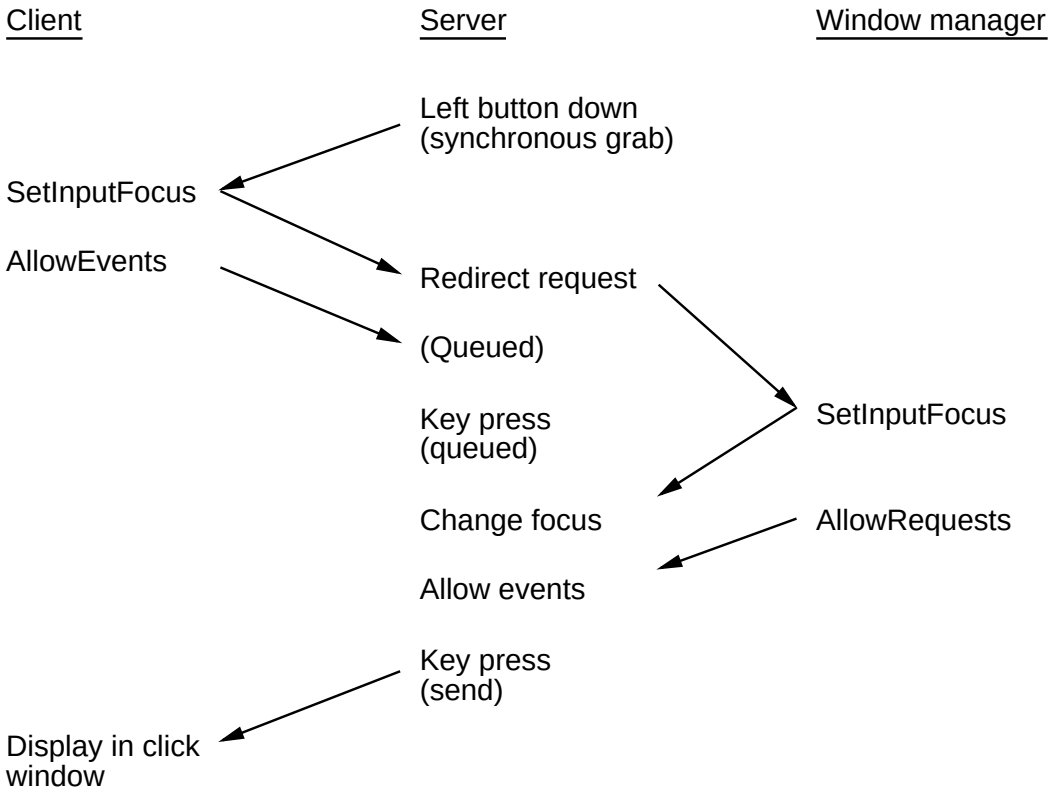

Figure 3-12: Synchronous redirection completely solves the PointerRoot focus race

either when the mouse leaves a top-level window, or when it enters a top-level window. Allowing synchronous grabs on `LeaveNotify` and `EnterNotify` offers the simplest solution. The window manager can install such grabs on each top-level window, and call `AllowEvents` after changing focus.

Recommendation: `SetInputFocus` should be a redirectable request, and a corresponding `FocusRequest` event should be added. For similar reasons, `InstallColormap` should be redirectable, and a `ColormapRequest` event should be added.[5] Synchronous grabs (or a similar event-freezing mechanism) should be extended to `EnterNotify` and `LeaveNotify`.

3.5. Window manager races solved with timestamps

The window managers described in the previous section strive for perfection—each user action, no matter how transitory, is treated correctly. But redirection takes time. While this isn't a problem when running the window manager locally on a fast workstation, we envision possible performance problems with X terminals where all clients, including the window manager, run remotely.

A window manager might wish to trade perfection for speed by renouncing the use of redirection. Such a window manager may improperly handle transient behavior, but should arrive

[5]Again, the ICCCM attempts to resolve this type of problem by legislating conventions that clients must adhere to. Our proposals provide a more powerful and concise solution.

in the desired end state soon after the user has stopped typing and mousing around. Even this modest goal is not achievable in X11.

Assume that a window manager wants to maintain input focus in a viewable window; if an application unmaps the current focus window U, the window manager heuristically chooses another window M to get focus. (The window manager keeps track of the current focus owner by tracking `FocusIn` events.) Suppose that the user clicks in yet another window F to give it focus after window U is unmapped, but before the window manager has reassigned focus to M (figure 3-13). The window manager has no way of knowing which timestamp to use in its `SetInputFocus` call, so it uses `CurrentTime`. Input focus ends up in window M rather than the user's chosen window F.

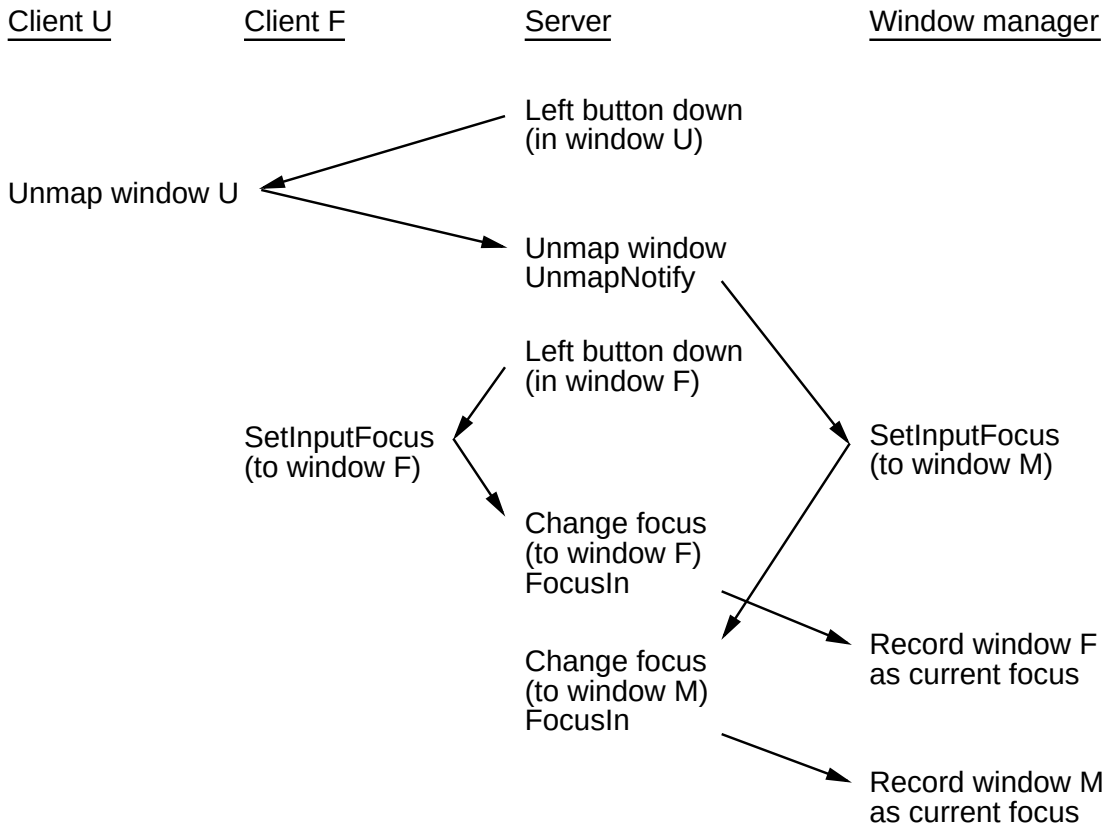

Figure 3-13: Unmap focus race

The window manager can avoid the race if it can use a valid timestamp in the `SetInputFocus` call. But where should such a timestamp come from? Ideally, the client would supply the timestamp from the first `ButtonPress` event to the `UnmapWindow` request, and the server would propagate this timestamp to the window manager in the `UnmapNotify` event. But in the current protocol, the `UnmapWindow` request doesn't take a timestamp parameter, nor does the `UnmapNotify` event contain a timestamp field. Adding timestamp fields to the appropriate window management notification events doesn't require recoding clients, but adding timestamp parameters to window management requests does.

A more compatible solution is to add a timestamp field to the `FocusIn` event, and report

FocusIn whenever that timestamp changes. This timestamp should be filled with the last-focus-change time. If an explicit SetInputFocus call changes the focus, the resulting FocusIn events contain the timestamp used in the call. If the server automatically changes focus because the focus window becomes unviewable, the last-focus-change time is not affected, so the resulting FocusIn events contain the timestamp of the last successful SetInputFocus call. Symmetry arguments strongly suggest making a similar change to FocusOut events, though we can't think of an example to justify it.

The window manager, which is already monitoring FocusIn events, would maintain a focus timestamp for each top-level window. Each time the window manager receives a FocusIn event on some window F, it copies the event timestamp into the focus timestamp for F. When the window manager reassigns focus because a client unmaps what the window manager believes is the current focus window, it uses the relatively old focus timestamp from the window *being unmapped*. This SetInputFocus call succeeds only if another client has not already changed input focus to another window, because the server accepts SetInputFocus requests with timestamps matching the last focus-change time.

Figure 3-14 shows how this works. If we redraw this figure so that the window manager's SetInputFocus arrives first, focus still ends up in the desired window F. Client F's SetInputFocus request will succeed, because its request will have a later timestamp than the window manager's timestamp U.

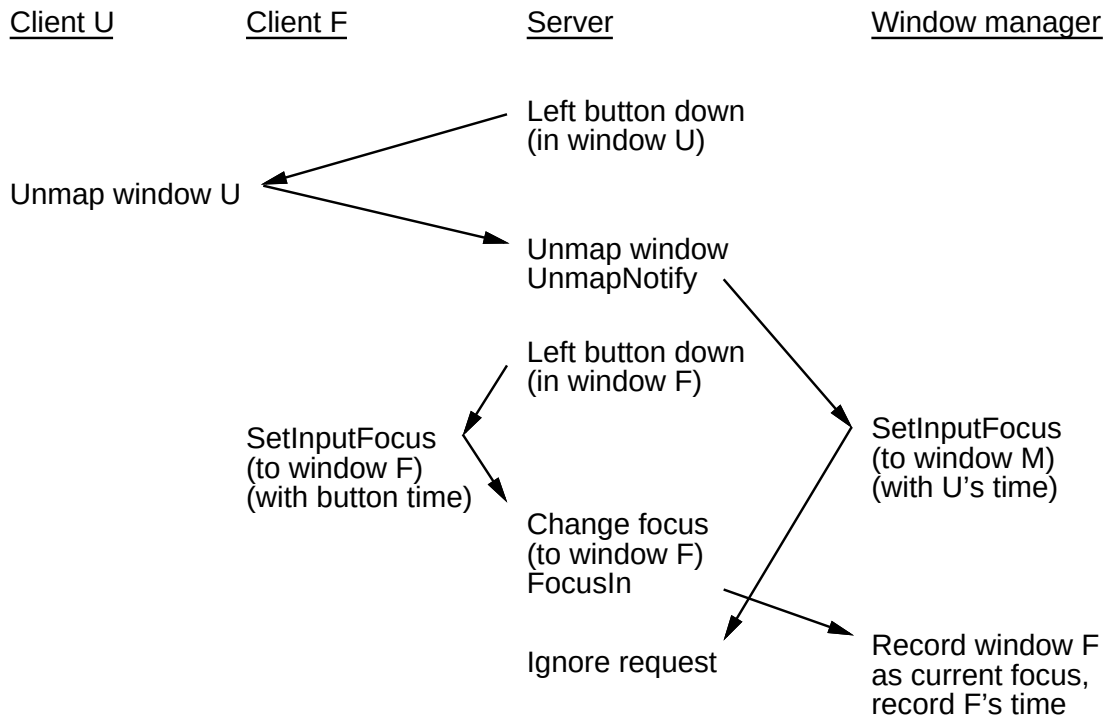

Figure 3-14: Timestamps solve the unmap focus race

Note that colormaps are subject to the same race conditions, and that these races can be solved by generating a timestamp for ColormapNotify events, assuming that InstallColormap takes a timestamp. Again, we believe that a protocol change offers a

more uniform model than a convention in the ICCCM.

Recommendation: The events `FocusIn`, `FocusOut`, and `ColormapNotify` should contain server-generated timestamps. In all cases the timestamp values should be the timestamp recorded when the resource was last set explicitly. The server should issue `FocusIn` events whenever the last-focus-change time is modified, even if the focus window remains the same. Similarly, additional `ColormapNotify` events must be issued when the timestamp of a colormap is advanced.

3.6. Other problems solved with redirection

The examples above show why all resource management requests, and most window management requests, should be redirectable. We now argue that the remaining window management requests—`UnmapWindow`, `DestroyWindow`, and `ReparentWindow`—should also be redirectable.

Currently none of these window management requests are redirectable, so window managers find out about the changed status of a window only after the fact. In the meantime, the window manager may have issued requests to the window that can no longer be applied, since no requests apply to a non-existent window, and many requests don't apply to unviewable windows. These operations eventually return an error to the window manager. A window manager might carefully record each request and why it could fail. It might verify that the appropriate unmap and destroy notifications have been received after issuing the request and before receiving the error. More likely, it will supply an error handler that blithely ignores all errors. While this last strategy usually works, it means that potentially serious bugs in window managers are likely to be masked.

Further, a client program might try to destroy, reparent, or unmap one of the windows owned by the window manager itself. If the client uses `QueryTree` to get information about the window hierarchy, and due to a programming error manipulates a window that the client doesn't own, the window manager can get very confused. It would be better to give the window manager the ability to monitor such requests before they are executed, rather than have it try to recover afterwards.

Finally, if `SetInputFocus` is made redirectable, race conditions can still arise unless `UnmapWindow`, `DestroyWindow`, and `ReparentWindow` are redirectable also. Let us return to the input focus race of the previous section, but with a window manager that redirects `SetInputFocus` requests to track changes in keyboard focus. This works fine as long as focus is changed explicitly by `SetInputFocus` requests. However, if an application unmaps, destroys, or reparents the current focus window, focus automatically reverts to the parent window, `PointerRoot`, or `None`. Since the server changes focus as a side-effect of unmapping, there is no explicit `SetInputFocus` request to allow the window manager to maintain control of focus.

If `SetInputFocus` were redirectable, the server could redirect the implicit focus change that may happen on `UnmapWindow`, `DestroyWindow`, and `ReparentWindow`, but this solution is extremely complex to implement and works only if redirection is synchronous. It is better to make the remaining window management requests redirectable.

It might seem odd to allow `DestroyWindow` to be redirected. After all, frequently a client

destroys its top-level window, then immediately terminates, breaking its connection to the X server. The connection loss causes the X server to recover the client's resources, including all of its windows. We see this situation as no different from any other in which a client breaks its connection, either deliberately or because of a programming error. A window manager must always be prepared to deal with this condition. `DestroyWindow` should be redirectable to simplify error handling in window managers, to prevent clients from destroying frame windows, and to avoid a tractable class of race conditions. (In this paper we do not propose any solutions to race conditions caused by connection loss.)

Recommendation: All window and resource management requests should be redirectable. In particular, `UnmapWindow`, `DestroyWindow`, and `ReparentWindow` should be made redirectable, and the corresponding redirection events `UnmapRequest`, `DestroyRequest`, and `ReparentRequest` should be added.

4. Tracking Window Attributes

Besides race conditions, there are two other problems that pose difficulties for window managers: some window attributes are not readable, and there is no notification mechanism for tracking changes to most window attributes.

4.1. Unreadable window attributes

Window managers that want to match their decorations (frames and icons) to the border or background of a client window cannot do so. The `GetWindowAttributes` request returns only a subset of the attributes that can be set with `CreateWindow` or `SetWindowAttributes`. In particular, the window manager has no way of determining a window's background color, background pixmap, border color, border pixmap, or cursor.

We know of no reason why the background and border pixels are not returned. Given these pixels, the window manager could match decorations to the window in most cases. Using `ColormapNotify` events (or redirection on `InstallColormap`), it could keep the decoration colormap consistent with the client window's colormap. And the window manager could use the client's colormap to determine exact RGB values for these pixels when coloring icons.

There are good reasons why the protocol doesn't return pixmap and cursor window attributes. The server may not be able to return the original pixmap handle, since the client may have already destroyed it. The server could return a copy of the pixmap, but this would waste memory. The server might sometimes do one or the other, but that leaves the client not knowing whether the resource should be freed.

A simple solution would be to permit `CreateWindow` and `SetWindowAttributes` calls to set the borders and backgrounds of one window to match those of another window. Note that we might want the border of one window to match the background of another window, and vice-versa. If the source window's background is `ParentRelative`, the server uses the background of the source window's parent.

This solution has none of the sharing problems described above. Even better, the `CreateWindow` request directly specifies the window to inherit attributes from, so the window

manager can avoid making a round-trip call to `GetWindowAttributes` to fetch the background or border.

Similar arguments extend to cursors. If the source cursor is `None`, the server uses the parent's cursor.

Recommendation: `GetWindowAttributes` should return a window's border and background color. `CreateWindow` and `SetWindowAttributes` should be extended to allow inheritance of border, background, and cursor from another window. It should be possible to cross-inherit border to background and background to border.

4.2. Untrackable window attributes

There are some window attributes whose values a window manager might want to track, but the X protocol provides no notification mechanism for tracking them. In the example above, the window manager may want to change decoration colors if a window changes its background color.

A more serious problem occurs in tracking changes to the save-under attribute of a pop-up dialog box window. Most applications set the value of the save-under field before mapping the pop-up window, and then never change it, so a reparenting window manager can propagate save-under when it creates its frame window. However, some applications might pop up a dialog box in different locations, and set save-under based on the difficulty of painting the underlying windows. In such cases the window manager must reliably track changes in the save-under attribute.

The most serious, albeit contrived, problem stems from the inability to track the override-redirect attribute, which if `True` prohibits requests from being redirected to the window manager. Suppose a client creates a window with override-redirect `False`, and a reparenting window manager decorates the window. The client maps the window, changes override-redirect to `True`, and resizes the window. The resize request is not redirected to the window manager, which consequently does not find out about the resize until the server sends a `ConfigureNotify` event.

To track such attributes, the window manager could poll windows occasionally, but this seems a poor solution even aside from efficiency considerations. For attributes like background pixmap, there is no way to poll. For the save-under attribute, the window manager can poll and risk mapping the window with the wrong value, or it can call `GetAttributes` before it maps the window, increasing the time it takes to pop up the window. For the override-redirect attribute, the problem described above remains if the client changes override-redirect to `True` and resizes the window between polls.

The protocol provides two mechanisms to track geometry-related changes to a window: a window manager can redirect `ConfigureWindow`, or can receive `ConfigureNotify` events. Both mechanisms should be extended by making `SetWindowAttributes` a redirectable request, and by adding an `AttributeNotify` event.

Again, backgrounds of `ParentRelative` and cursors of `None` require special attention. Any time the server changes a window's background, it should (recursively) propagate redirection

or notification of this change to any children with a background of `ParentRelative`. When the server changes a window's cursor, it should propagate redirection or notification to any children with a cursor of `None`. Note that redirection in this case does not prevent the change to the parent from taking place, but serves more as a notification mechanism. This is stretching the notion of redirection a bit, but it seems better than having to ask for both redirection and notification in order to track background and cursor changes.

Recommendation: `SetWindowAttributes` should be a redirectable request, and the corresponding `SetAttributesRequest` event should be added. If the window manager asks for redirection of `SetWindowAttributes`, the server redirects any request that changes the override-redirect attribute, regardless of the current state of the override-redirect attribute. An `AttributeNotify` event, analogous to other notify events, should be added to the protocol.

5. Window Visibility

Clients generally use windows in one of two ways. Most windows are like sleeping dogs—when the user pokes them, they jump up, roll over, and after a bit of frenzied activity, play dead. Real-time windows display data that are constantly or periodically updated; simple examples are clocks and graphs of machine loads. Here we are concerned with real-time windows whose contents are expensive to compute or display, like frequency analyzers, video movies, and image processing programs.

To reduce load on the host machine and the server, real-time windows want to avoid painting bits that are not visible to the user. If the entire window is invisible, the application can probably avoid some computation and all painting. If the window is partially visible, the application may be able to avoid some of its computation and painting calls. Such an application needs accurate information about what regions of the window, if any, are visible.

The X protocol fails in two respects: a window can become invisible without generating any notification to the client, and in the cases where the client does receive notification there is too little information.

5.1. VisibilityNotify for unviewable windows

The X protocol carefully defines the terms *viewable* and *visible*. A window is viewable if it and all of its ancestors are mapped. A window is visible if some portion is actually visible to a user. A window must be viewable to be visible. However, a viewable window might not be visible—it can be completely obscured by windows above it.

Clients ask for `VisibilityNotify` events to find out about changes in the visibility of a window. `VisibilityNotify` specifies whether the window is fully visible, fully obscured, or partially visible. The server generates a `VisibilityNotify` event when a window goes from not visible to partially or fully visible. It also generates `VisibilityNotify` when a window goes from unviewable to viewable, even if the window remains not visible. However, the server doesn't generate `VisibilityNotify` when a window goes from viewable (and possibly visible) to unviewable (and definitely not visible).

If a client knows that a real-time window is viewable, it can combine this information with `VisibilityNotify` events to determine the true visibility of the window. To know whether a

window is viewable, a client must know if the window and all of its ancestors are mapped. The client code responsible for the window is usually a module that has no direct knowledge about the window's ancestors; the module must query the server to discover the window's lineage, then look for `UnmapNotify` events on the window and all of its ancestors. Further, since reparenting window managers can change the shape of the window tree, this module must periodically poll the server to see if the window hierarchy has changed. Rather than burden clients with this task, the server should provide viewability information in the `VisibilityNotify` event.

Recommendation: `VisibilityNotify` events provide a detail of `Unobscured`, `PartiallyObscured`, or `FullyObscured` to describe the state of a window that is viewable. This detail should be extended with an additional state, `Unviewable`, which is generated whenever the window changes state from viewable (regardless of visibility) to unviewable.

5.2. Partially obscured windows

The above extension to `VisibilityNotify` allows clients to reduce computation if the window is not visible at all. But some real-time clients would like to know exactly which regions of a window are visible in order to further optimize performance. This information is difficult to come by. The server provides no notification when new portions of a partially obscured window become obscured, and it provides incomplete notification when new portions become unobscured.

A client can determine which portions of a real-time window are visible by occasionally polling the server for the window tree and computing window clipping itself. This is cumbersome, but should suffice for discovering that portions of window have become obscured since the last poll. Since the window needs this information only for efficiency reasons, prompt notification is not essential.

On the other hand, clients want to paint newly visible portions of a window promptly. The protocol does not offer a means to do this with acceptable performance. Although polling would allow a client to discover visibility changes, polling often enough to provide good response would be prohibitively expensive. The client might try to use `Expose` events, but `Expose` events aren't generated for windows with backing store. Turning off backing store doesn't work either because the protocol permits the server to provide backing store even to clients that don't ask for it. Besides, clients for whom painting is computationally expensive are exactly those clients that most benefit from backing store.

Recommendations: A new event mask `BackingExposure` should be added. If `BackingExposure` is selected on a window, the server generates `Expose` events even if the window has backing store. The `Expose` event should include a new field `backing-expose` which is `True` if the region described by the event is repainted from backing store. A new event mask `Unexposure`, and the corresponding event `Unexpose` should be added. If `Unexposure` is selected on a window, the server generates a series of `Unexpose` events (similar in structure and semantics to `Expose` events) whenever a portion of the window becomes obscured.

6. Mouse Tracking

Simply displaying the mouse cursor is the server's problem. In many cases, hardware makes this job trivial. However, clients frequently provide specialized mouse-tracking feedback: rubberband lines and boxes show the size of a line or rectangle to be created, crosshairs indicate mouse position on rulers surrounding the window, inverted or underlined text shows the extent of a text selection.

Effective mouse tracking should satisfy two possibly conflicting goals: the display should track changes in mouse position as quickly as possible, and the client should not create an excessive load on the network or on the client and server machines. Further, the client can make no assumptions about the relative efficiency of the server's painting operations and the communication link to the server. Painting may be insignificant compared to the time for a round trip or the time between two mouse motion events, or it may be much larger than these times. We assume it is trivial for the client to issue painting requests in response to `MotionNotify` events.

In the sections below we analyze three methods that X clients use for mouse tracking, and point out problems with each. We finally suggest a protocol change that would allow efficient, correct, and reasonably interactive mouse tracking.

We analyze interactiveness by computing mouse-update times. These are the minimum and maximum times that can elapse between mouse movement and the completion of painting requests to reflect this or a later position of the mouse. The mouse-update times are expressed in terms of three parameters: M, the mouse-sampling period;[6] RT, the average round-trip time between server and client; and P, the painting time to update the tracking feedback. Smaller maximum mouse-update times mean more interactive mouse feedback; smaller differences between the minimum and maximum time mean less variability in mouse feedback. Correctness means that for any combination of M, RT, and P, the maximum mouse-update time is a finite number. Efficiency means that if the mouse isn't moving, the server and client aren't wasting cycles and network bandwidth talking about it.

6.1. Asynchronous tracking

The simplest way for a client to track the mouse is to ask for `MotionNotify` events, and to issue painting requests in response to each motion event. This works quite well if the painting time is small. The minimum mouse-update time is one round-trip time plus one painting time (1 RT + 1 P). The maximum mouse-update time is one mouse-sampling time plus one round-trip time plus one painting time (1 M + 1 RT + 1 P). This is the best we can hope to do.

However, the computation above of the maximum mouse-update time is flawed if the painting time is greater than the mouse-sampling time. If the user moves the mouse continually, then the client generates painting requests faster than the server can paint them. The mouse tracking feedback falls farther and farther behind the mouse. Thus, if P > M, the maximum mouse-update time is unbounded.

---

[6] The protocol specifies no minimum granularity for the mouse-sampling period, nor does it require that the mouse be sampled at regular intervals. In practice, servers sample the mouse at regular intervals ranging from about 1/100 to 1/60 of a second.

Some clients compress `MotionNotify` events in an attempt to avoid this and other tracking problems.[7] If a client sees more than one motion event in its input queue, it discards all but the last one and generates a single painting update. But the protocol doesn't specify what to do first if a server has events to deliver to a client, and at the same time has painting requests outstanding from the same client. Giving priority to painting lets real-time clients indefinitely lock out their own events, so servers invariably give priority to delivering events. As a result, clients often see only one `MotionNotify` event at a time, rendering client-side compression useless.

Further, *any* specification of priorities, no matter how complex, will always be susceptible either to locking out events to a real-time client, or to lagging increasingly far behind the mouse. Some synchronization between motion events and paint requests is therefore necessary.

Synchronization need not require a round trip. For example, a mouse-tracking marker request could be added to the protocol. A client sends this request immediately after it issues the painting requests to update mouse tracking feedback. The server always sends `MotionNotify` events immediately if there are no mouse-tracking markers in the queue. Otherwise the server processes requests until there are no mouse-tracking markers in its input queue, and then sends all queued events out at once. This solution requires the server to look ahead for markers in its incoming request stream, which no other protocol request requires.

6.2. Synchronous polling

In another popular tracking method, the client polls the server for the mouse position using `QueryPointer`, then paints. Repeat ad infinitum. The minimum mouse-update time is 1 RT + 1 P. The maximum time occurs when the mouse moves immediately after the server sends the results of a `QueryPointer`. The new (or later) mouse position won't be painted until the client receives the results from `QueryPointer`, issues painting requests, queries the mouse position again, and issues painting requests for that position. This gives a theoretical maximum mouse-update time of 2 RT + 2 P. Since most servers don't actually poll the mouse in `QueryPointer`, but use the slightly stale position from the last mouse-sampling time, the maximum time is 1 M + 2 RT + 2 P.

If the round-trip and painting times are small enough, synchronous polling becomes busy waiting. The client polls the server multiple times between mouse-sampling times, getting the same mouse position back each time. The client and server saturate all available cycles, and may also consume a substantial portion of the network bandwidth. Polling is less efficient and usually slower than asynchronous tracking. Its sole advantage is that it has a bounded mouse-update time, so it never falls farther and farther behind the mouse.

6.3. PointerMotionHint tracking

The X11 protocol allows clients to ask the server to filter `MotionNotify` events by selecting for motion events with `PointerMotionHint`. If `PointerMotionHint` is selected, the protocol suggests that the server should deliver a single `MotionNotify` event under certain conditions. The application can then use the `QueryPointer` request to get the current mouse

[7] In fact, the Xtk toolkit compresses `MotionNotify` events for clients that request it.

position and tell the server to send another MotionNotify event when the conditions are again satisfied.

While it is intended to help with mouse tracking, PointerMotionHint is unusable in practice. In particular:

1. PointerMotionHint is only a hint and may be ignored: the server is not required to filter any MotionNotify events. In this case, the busy wait problem returns.
2. Even if a server implements PointerMotionHint filtering, the protocol does not specify exactly when the server should send a MotionNotify event after the QueryPointer request. The server is free to send a MotionNotify event immediately, which causes busy waiting. Alternatively, the server is free to delay reporting that the mouse has moved, which causes tracking to stop even though the user is moving the mouse. This is even worse than the unbounded asynchronous case above, because the tracking feedback may never catch up with the mouse position after the user stops moving the mouse.
3. It is difficult to process mouse positions in the correct context using PointerMotionHint. Since QueryPointer is a reply request, the coordinates it reports are seen as a reply value and not a part of the ordinary event stream. Thus, it is easy for the client to inadvertently ignore intervening key and button events.

6.4. Lazy polling

We now present a better way of updating mouse tracking which we call *lazy polling*. Lazy polling provides the capabilities for which PointerMotionHint was intended, but is easier to use and implement, and results in better minimum mouse-update times.

Lazy polling requires a new event-selection mask, PointerMotionAllow, and a new protocol request, AllowMotion, which takes a window and a position as arguments. If PointerMotionAllow is selected, the server sends at most one MotionNotify event after receiving an AllowMotion request. If the mouse is no longer at the specified position, the server immediately sends a MotionNotify event reporting the current position. If the mouse is still at the specified position, the server sends a MotionNotify event as soon as the mouse moves.

In this way, applications can simply accept MotionNotify events, call AllowMotion to allow the next interesting MotionNotify event, and update the tracking feedback. (Requesting AllowMotion before updating the feedback improves average response time over networks where RT is relatively large.) The minimum mouse-update time is 1 RT + 1 P; the maximum mouse-update time is 1 M + 2 RT + 2 P.

Recommendation: PointerMotionHint is so loosely specified and difficult to use correctly that we recommend eliminating it from the protocol. The protocol should replace the selection mask PointerMotionHint with PointerMotionAllow. A new request, AllowMotion, sends one MotionNotify event when the mouse has moved to a position different from the one specified.

7. Pop-up and Redisplay Efficiency

The X protocol's network-transparent client-server model is less efficient than a direct-procedure-call window system, such as SunView [8] or Microsoft Windows [10]. In particular,

1. Requests and events must be packaged and unpackaged.
2. Request and event packets must be moved across address spaces or across a network.
3. A context switch is required to move between executing client code and server code, if they are running on the same machine.

Most of the time, clients are sending a stream of requests and receiving a stream of events, allowing some costs due to the second and third points above to be amortized over large packets. The remaining overhead usually isn't a significant problem: servers spend more time painting than sending events and receiving requests, and local-area networks provide adequate transport.

However, synchronous round trips, which are required in window management, cannot amortize the above costs and so are inherently more expensive. Furthermore, in situations requiring fast reponse, such as popping up a menu, network bandwidth and latency are often inadequate. This is particularly a problem when X is used outside of its design parameters, for example using phone lines in place of a network. Just sending the `Expose` events over a phone line for a typical pop-up dialog box can take several seconds.

7.1. Popping up menus

Menus are the simplest pop-up: they do not set input focus, and they are invariably created with override-redirect `True`, so the window manager never sees them. Popping up a menu requires two round-trip times, plus processing of several `Expose` events: the server sends a `ButtonPress` event, and the client responds with a `MapWindow` request. Then the server generates a series of `Expose` events—one for each visible subwindow in the menu—and the client responds with painting requests for each window.

Waiting for the `Expose` events is inefficient; clients could start painting immediately after issuing the map request and ignore the forthcoming `Expose` events. However, this introduces added complexity if the clients want to avoid repainting the menu when they finally receive the `Expose` events generated by the map request. Furthermore, the cost of generating, packaging, transporting, and unpackaging the `Expose` events still remains, even if the client chooses to ignore them.

Our solution is to parameterize `MapWindow` and `MapSubwindow` requests to indicate whether the server should generate `Expose` events caused by the map. This completely solves the performance problems for override-redirect windows.

Recommendation: `MapWindow` and `MapSubwindows` should take an `exposures` parameter as the `ClearArea` request does. If exposures is `False`, the requests generate no `Expose` events.

7.2. Popping up dialog boxes

Dialog boxes are more complex than menus. Many dialog boxes are created with override-redirect `False`, so a window manager may redirect the `MapWindow` request. Dialog boxes often set the input focus to some window in the dialog box; if `SetInputFocus` is made redirectable, a window manager may redirect it as well.

Asynchronous redirection doesn't just complicate clients; it makes them inefficient as well. Because a client must operate correctly under a redirecting window manager, it cannot use the same shortcuts for dialog boxes that it can for menus. It must instead wait for various notification events before issuing certain requests. The *possibility* of redirection adds one round trip; the actual *use* of redirection by a window manager adds another. Figure 7-1 illustrates both scenarios: dotted lines show the extra round trip if the client runs under a redirecting window manager.

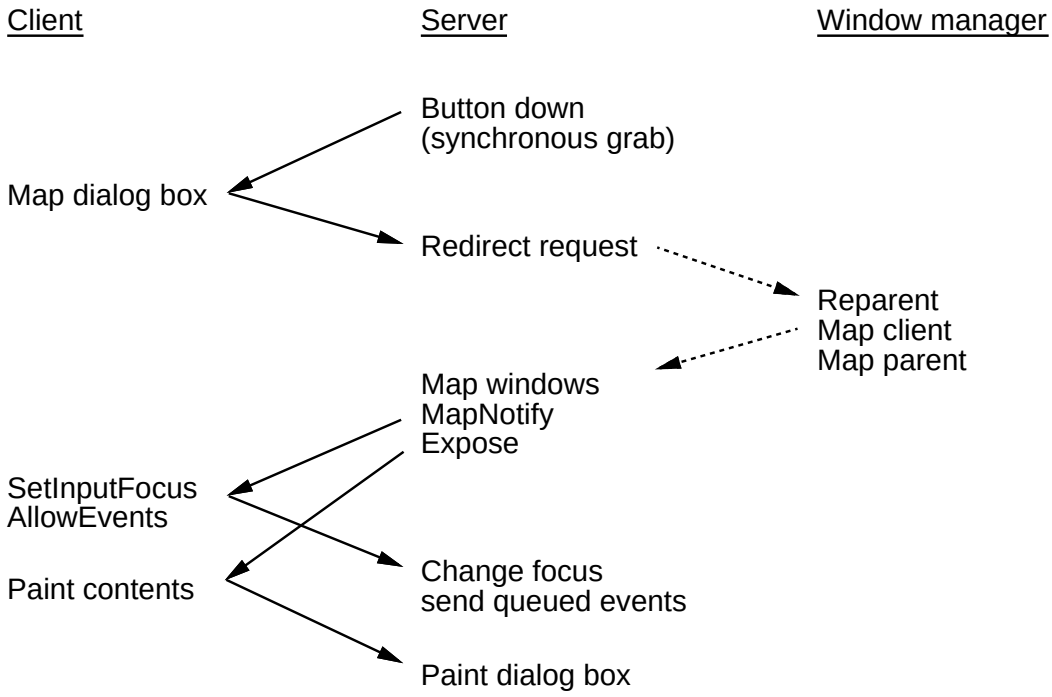

Figure 7-1: Popping up a dialog box slowly

Efficiency improves considerably if redirection is synchronous and the client can map a window without generating `Expose` events. The client responds to the `ButtonPress` event by issuing `MapWindow`, `SetInputFocus`, `AllowEvents`, and painting requests. This reduces dialog box pop-up to one round-trip time if the window manager is non-redirecting, and to two round-trip times if the window manager redirects `MapWindow`, as shown in figure 7-2.

Even in the presence of synchronous redirection, the decision to generate painting requests immediately requires some justification, because a redirecting window manager might resize the dialog box before mapping it, or might not map the dialog box as the topmost window. Waiting for `Expose` events might appear to be better than writing to a window that could be the wrong size or partially obscured. In practice, window managers don't change the geometry of dialog boxes; even if they did, painting immediately would work well if the client uses window and bit gravity

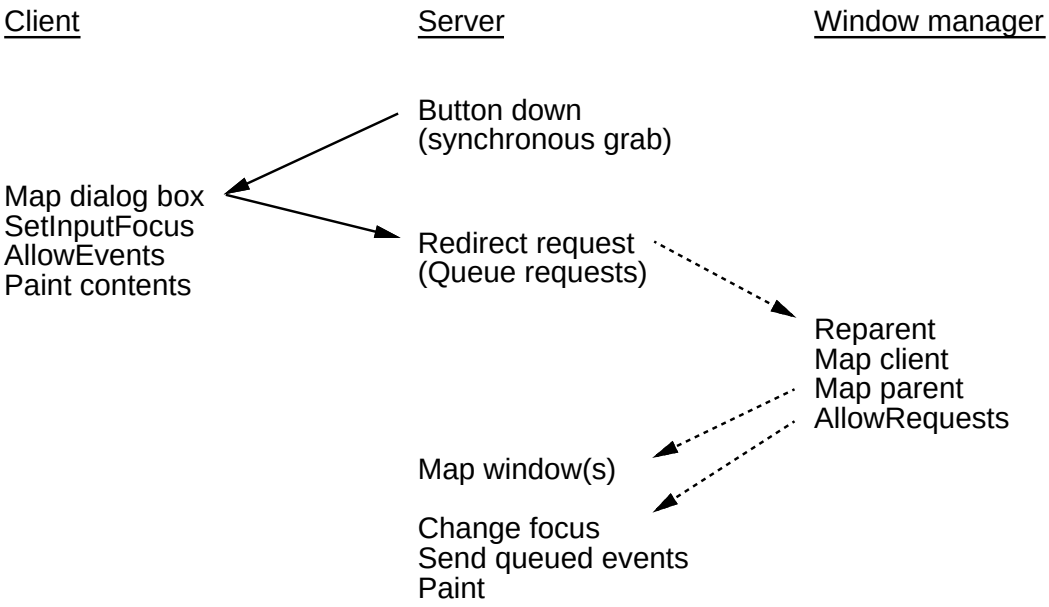

Figure 7-2: Popping up a dialog box quickly

sensibly.

If a window manager uses synchronous redirection for `MapWindow`, it might redirect `SetInputFocus` as well (if `SetInputFocus` is made redirectable). In this case, popping up the dialog box with synchronous redirection requires three round trips, the same as the existing asynchronous redirection. As soon as the window manager calls `AllowRequests` after mapping the dialog box, the server redirects `SetInputFocus` and again freezes processing of client requests.

We can remove the extra round trip entirely by slightly altering the definition of synchronous redirection. Redirection should not freeze *all* subsequent client requests; it should freeze the client only upon receiving a request that is not redirected, or a request that is redirected to a different destination. The server increments a client redirection counter each time it redirects an event, and decrements the counter each time it receives an `AllowRequests` call for the client. Normal request processing in the client does not resume until the redirection counter reaches 0.

This way, the server redirects `MapWindow`, then immediately redirects `SetInputFocus` without waiting for the window manager to call `AllowRequests`. The server resumes client request processing upon receiving two `AllowRequests` calls—one for the `MapWindow`, and one for the `SetInputFocus`.

This redefinition of synchronous redirection maintains the guarantee that client requests are not reordered. However, window managers must deal with the possibility of receiving a second redirected request before receiving event notification from their response to the first redirected request. In the example above, the window manager receives the redirected `SetInputFocus` request before any `MapNotify` events. We are willing to trade a small increase in window manager complexity for faster pop-ups.

Recommendation: When handling synchronous redirection, the server should freeze client

request processing when it encounters a request that is not redirected or a request that is redirected to a different destination. The server should count the number of requests redirected before freezing, and should resume normal client event processing when it receives a corresponding number of AllowEvents requests.

7.3. Expose event grabbing

Whenever a large section of a window gets exposed, all of the subwindows inside it get Expose events. For windows with complex internal structure, the number of such Expose events could be very large. Moreover, for many applications, the information contained in the Expose events could be computed by the client from a single Expose event on the parent.

For example, the Xtk toolkit shadows window positions and sizes in order to speed up the negotiations for geometry layout between parent windows and their children. Xtk has enough information to select for Expose events on the top-level client window, and then internally generate calls to the proper subwindow exposure procedures. However, there is no way to request that Expose events on children be coalesced and reported with respect to an ancestor.

Grabs provide a model for extending Expose semantics. Besides using grabs for mouse-ahead and type-ahead, a client can use them to alter the normal flow of event processing in the server. The server normally processes a mouse or keyboard event in a bottom-up fashion; the lowest containing window that has expressed interest gets the event. Grabs use top-down processing, and thus let a higher-level window intercept the event. The grabbing client can swallow the event completely, or can replay the event to the next lower level in the window hierarchy.

If a toolkit could grab Expose events on the top-level window of a client, then it could distribute this information to the proper exposure procedures directly. At worst, the toolkit would perform approximately the same amount of work to call exposure procedures directly as the server performs in generating the information contained in the Expose events. In any case, Expose grabs eliminate the time to package, send, and unpackage tens or hundreds of events. This would be particularly useful over serial links.

Recommendation: Grabs should be extended to Expose events, and servers should coalesce all exposures on subwindows into Expose events on the grabbed window. If Unexpose events are added, grabs should be extended to these events as well.

8. Exceptional Conditions

The protocol isn't helpful about getting users out of bad situations. The most drastic errors lock up the server entirely, and regaining control requires using another machine or rebooting. Less serious, though still annoying, problems occur when users try to abort painting requests; an application can offer better abort response only at the risk of slowing down the normal case.

8.1. Things that go grab in the night

If an X11 client issues a grab using GrabPointer, GrabKeyboard, or GrabServer and fails to release the grab, all other clients, including the window manager, are locked out. If a client establishes a synchronous passive grab using GrabButton or GrabKey and fails to call

`AllowEvents`, the grab technically ends when the button or key is released, but the server never interprets and dispatches user events. In both cases, the only way to recover control of the display is to use another machine to kill the offending program, or to reboot.

A client might fail to release a grab due to incorrect grabbing code. For example, the client may wait to release a grab until it receives a certain event, but there are situations in which the server never generates the event. More likely, a client program has a bug unrelated to the grab, and the user runs the client under a debugger. In trying to elicit various behaviors, the user may click in a client window that establishes a synchronous passive grab while the debugger has the client paused. Mouse and keyboard event processing freezes, and the user loses control.

The situation in the Sprite operating system [5] is even worse. When a program crashes, Sprite doesn't dump core, but leaves the program halted under a special debugging process. This makes much more context available than post-mortem debuggers provide—for example, the user can see the contents of the X windows that the program owns. Of course, these windows show no indication that the program behind them is halted. If the user clicks in a dead window that has registered a passive grab, everything freezes.

One solution is to establish a maximum time that a client can hold a grab. If the client doesn't release the grab or allow events in that time, then the server automatically breaks the grab. But if a developer uses a work machine and a test machine to debug code that uses grabs, he will be frustrated if the server on the test machine takes away the grab he is trying to debug. At the very least, a grab timeout should be a settable resource like the screen saver timeout.

A different solution is to allow an escape mechanism. A client (which would usually be the window manager) should be able to register a key combination that breaks all grabs, thaws all frozen devices, and resets focus to `PointerRoot`. Using this solution, the user neither has to wait for the timeout period nor change the timeout duration when cross-machine debugging. This solution makes it possible to debug most errors with grabs on a single workstation; we hope that this will help get the bugs out, so that only application developers will ever need to use this facility.

Recommendation: A new protocol request, `SetBreakGrabs`, allows a client to register a key combination which is very difficult to type. When typed, this combination breaks all existing grabs and unfreezes the server.

8.2. Aborting queued requests

Some terminal-based programs are quite reactive to user input; the *emacs* text editor is probably the best-known example. When the user issues commands to move to a new page, *emacs* stops sending characters from the current page, and starts sending the new page as soon as possible.

X clients cannot efficiently provide similar functionality. Requests like `PolySegment` or `PolyArc` can take a long time to paint if the graphics context specifies a wide line. Since these requests take a relatively small amount of data, the server can queue up a large number of them. Even if the client program stops sending painting requests as soon as the user indicates lack of interest in the current view, the queued requests continue to execute for several seconds.

To improve abort response, a client can break most complex painting commands into multiple smaller requests, and synchronize with the server after each request. Synchronization requires a round trip, which decreases performance. Achieving instantaneous abort response can easily double the time it takes to paint the entire window; application writers must settle for some compromise between responsiveness and efficiency.

We offer no solutions to this problem. A separate channel for communicating with the server would help, but this would be a large change with ramifications we are unprepared to explore. We note that such a channel could also be used to synchronize the asynchronous mouse tracking case without requiring the server to look ahead in the normal communication channel.

9. Conclusions

This paper provides comprehensive descriptions of the problems that we see in the X protocol. Its length should not be taken as an indictment of the X protocol, for in fact the protocol is extremely well designed. Some of the problems we present result from the overwhelming success of X11: it is being used in situations the designers never envisioned. Many of the performance problems could not have been anticipated, and will decrease in importance as faster X servers and hardware become available. Some of the problems can be coded around. Finally, as we sifted through possible solutions to problems, we were constantly impressed at the number of subtleties in the design that made our job easier.

In evaluating solutions, we focussed mostly on technical issues rather than political considerations; our solutions often look like proposals for X12 rather than fixups for X11. At first glance, this approach differs markedly from the philosophy behind the Inter-Client Communication Conventions Manual. The ICCCM doesn't change the X protocol, avoiding any need to change servers. It does prescribe a large set of conventions that clients must obey in order to work peacefully with one another. Complying with these prescriptions has required reprogramming existing clients and window managers, and many of the required changes have not been made correctly. Thus, upon closer inspection, changing the protocol might have provided a more effective means of avoiding races and imposing consistent focus and colormap management styles. An ICCCM would still be needed in the areas of selections and window manager hints.

Most of our protocol changes can be introduced in upward-compatible ways by offering new servers and window managers first, and then reprogramming or relinking clients at a more leisurely pace. For example, if a server sees the current 8-byte `InstallColormap` request, it assumes a timestamp of `CurrentTime`. But the server could also accept the new 12-byte `InstallColormap` request, which contains an explicit timestamp. Similarly, making redirection synchronous requires only that the server and window managers change simultaneously. Unmodified clients still work, and can be reprogrammed later to take advantage of the performance gains that synchronous redirection makes possible. Note also that synchronous redirection would allow protocol requests and events to be clearly separated into ''normal'' and ''window manager'' sections, and would eliminate the subtle scattered references to possible behavior of requests when override-redirect is `False`.

In some sense, this paper is too late—people writing X11 clients don't want to rewrite code, regardless of the benefits. Unfortunately, many of these problems did not arise until programmers tried to accomplish various tasks in X. The very popularity that has led to the

discovery of these problems may also prevent the adoption of reasonable solutions. At the very least, we hope the reader has gained an understanding of the problems inherent to the X11 protocol.

10. Acknowledgements

Robert Scheifler and Jim Gettys provided extensive clarification of fine points in the protocol, and were quite open about mistakes and omissions in the protocol.

References

1. Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1985.

2. Apple Computer, Inc.. *Inside Macintosh, Volumes I-III.* Addison-Wesley, Reading, Massachusetts, 1985.

3. James Gosling, David S.H.Rosenthal, and Michelle Arden. *The NeWS Book: An Introduction to the Networked Extensible Window System.* Springer-Verlag, 1989.

4. Joel McCormack, Paul Asente, Ralph Swick. *X Toolkit Library - C Language Interface.* X Version 11 Release 3 edition, Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA, 1988.

5. John Ousterhout. Private communication.

6. International Standards ISO/IEC 9592-1:1988(E). *Programmer's Hierarchical Interactive Graphics System (PHIGS).* International Standards Organization, Geneva, 1988.

7. Robert W. Scheifler, James Gettys. *X Window System: The Complete Reference to Xlib, Xprotocol, ICCCM, XLFD.* Digital Press, 1990.

8. Sun Microsystems, Inc. *SunView Programmer's Guide.* 4.1 edition, Sun Microsystems, Inc., Mountain View, California, 1990.

9. Tom Thompson. "The Next Step". *Byte Magazine* (March 1989).

10. Microsoft Corp. *Microsoft Windows Programmer's Reference.* Redmond, Washington, 1990.

Table of Contents

List of Figures