A Concurrent Window System

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

When implemented in a concurrent language, a window system can be concise. If its client programs connect to the window system using an interface defined in terms of communication on synchronous channels, much of the complexity of traditional event-based interfaces can be avoided. Once that interface is specified, complex interactive programs can be assembled, not monolithically, but rather by connecting, using the same techniques, small self-contained components that each implement or modify elements of the interface. In particular, the window system itself may be run recursively to implement subwindows for multiplexed applications such as multi-file text editors. This is the software tool approach applied to windows.

Introduction

Traditional window systems offer their clients — the programs that call upon them for services — an interface consisting primarily of a graphics library and a stream of 'events': tokens representing key presses, mouse motion and so on. The clients of these systems tend to be written as state machines with transitions triggered by these events. Although this style of programming is adequate, it is uncomfortable; state machines are powerful but inconvenient. There are also engineering reasons to object to this interface style: all types of events come through the same port and must be disentangled; the event handlers must relinquish control to the main loop after processing an event; and the externally imposed definition of events (such as whether depressing a mouse button is the same type of event as releasing one) affects the structure of the overall program.

To make the situation more comfortable, event-driven interfaces typically allow some extra control. For instance, a program displaying a pop-up menu can usually arrange to ask only for mouse events, so the code supporting the menu is not disrupted by keyboard events. Although they help, such details in the interface are just work-arounds for the fundamental difficulties that event-driven programming must ultimately face. The work-arounds accumulate: the X11 windows library, for instance, has 27 standard entry points to handle 33 types of events [Xlib 88]; NeWS has a single general event type but still needs 20 entry points to handle it [Sun 87]. The interface for input in GKS is comparably complex [GKS 84]. It is demonstrably difficult to write simple programs to connect to such intricate interfaces [Rose 88].

Although events from the various inputs may be intermingled and asynchronous, the events from any one device will be well-behaved. We could therefore program each device synchronously and cleanly if we could divide the events into separate streams, one per device, directed at concurrent processes. The resulting program would be a collection of self-contained processes, free of irrelevant bookkeeping, whose execution would be interleaved automatically.

That approach was taken in Squeak [Card 85], a concurrent language designed for programming such components of user interfaces as menus and scroll bars. Squeak was a small language, though, and too

---

NeWS is a trademark of Sun Microsystems, Inc.
PostScript is a registered trademark of Adobe Systems, Inc.
UNIX is a registered trademark of AT&T.

simple to be useful for writing complete applications. It lacked variables, a type system, communication of anything other than integers, and dynamic creation of processes. Input management in a realistic environment requires a stronger language than Squeak and a more concentrated understanding of graphics applications and the environment in which they run.

This paper describes the design of an experimental window system written in a concurrent language designed for the job. The language, called Newsqueak, is documented elsewhere [Pike 89]. The window system provides a well-specified environment for its client programs, using a synchronous procedural interface for output and structured communication on a small number of synchronous channels (as in CSP [Hoare 78]; see below) to handle input and control. The window system functions by multiplexing its clients' access to its own environment, which has the same structure, allowing the system to be run recursively. The environment is easy to program both from the client's and the window system's points of view. The complete window system is fewer than 300 lines of Newsqueak.

Comparison with other systems

In contrast to systems such as NeWS, which allow their clients to be written using concurrent techniques, this system *requires* its clients to be written concurrently. The only interface to the window system is through parallel synchronous communication channels. The main observation from this exercise is the importance of specifying the interface between the window system and its clients succinctly and completely. Given such a specification, it becomes possible to interconnect small programs, each of which implements a piece of the specification, to form larger interactive applications in a manner similar to the pipeline approach to text processing on the UNIX system.

The external structure of the system is akin to that of NeWS. Remotely executing applications implement graphical user interfaces by connecting to the window system over the network and calling upon the Newsqueak interpreter to execute code on their behalf. It is therefore expected that production applications would be ordinary compiled programs running remotely that achieve interactive graphics by loading customized Newsqueak code into the window system's connections to them.

Philosophically, the closest relative might be the Trestle window system of Manasse and Nelson [Mana 89]. There, the connection between the window system and its clients is implemented by a bidirectional module interface. By defining that interface thoroughly, Trestle achieves some of the same interconnectivity and recursive structure but in a more conventional environment. (From some points of view, object-oriented programming, modules, and concurrent communications on synchronous channels are the same idea in different clothing.) The input mechanism in Trestle is still event-driven, however. The system described here goes further, defining all input and window control functions, including resizing and deleting windows, using synchronous communications.

Basics

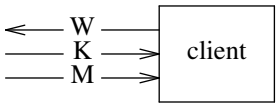
An application or *client* of a window system is an independently executing process whose display activity is confined to a subset of the entire screen controlled by the window system. That subset is the client's *window*. Ignore for the moment the process-specific aspects of the client and assume that it is a procedure written in a high-level programming language. Moreover, assume that there is no implicit global environment for the procedure. Instead, the environment for the procedure must be passed explicitly, as parameters, from the window system to the client. If we can define those parameters and what they signify, we will have completely defined the properties of a client.

One of the parameters will obviously be a variable, say `W`, labeling the window in which the program is running. `W` will be used by the client to place output in its window. `W`'s precise type need not concern us yet, but it must at least describe the window's geometry.

`W` is, loosely, a capability, granted by the window system, enabling the client to use its window. `W` labels a multiplexed component of a larger screen containing all the windows. Input to the client may be regarded similarly. We need capabilities allowing communication with the multiplexed mouse, keyboard, and perhaps other input devices. Call these `K` and `M`, and pass over their exact specifications. The client's declaration is then, approximately,

```
client: prog(W, K, M)
```

(The syntax used in this paper is based on Newsqueak, but should be mostly self-explanatory.) Once it has been invoked, the client can be represented pictorially, using arrows to represent the flow of information.

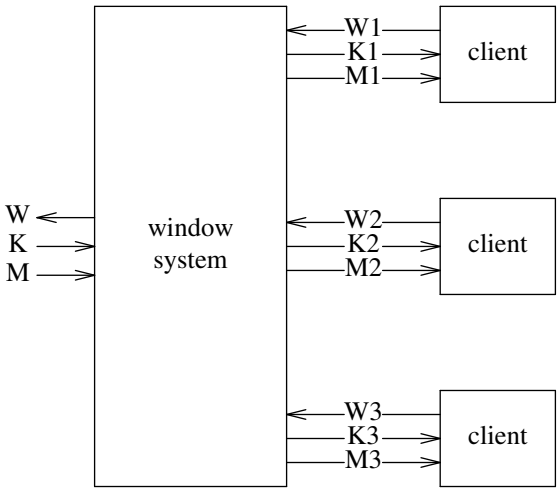


Ultimately the picture will become more complicated as we add new possibilities such as setting the mouse position and controlling the resizing and deleting of windows.

The window system has several independently executing clients, each of which has the same external specification. It multiplexes a screen, mouse, and keyboard for its clients and therefore has a type reminiscent of the clients themselves:

```
windowsys: prog(S, K, M)
```

where `S` is the screen. If we arrange the client's windows to be programmable by the same interface as the full screen (making `S` a `W`), the window system will have the same type as its clients. Pictorially,



If the window system multiplexes clients of its own type, then it may be a client of itself, or a client may pass its environment to a fresh invocation of the window system to do further multiplexing. This recursion allows a client of window system — a text editor, say — to invoke the window system afresh in its window to manage subwindows for multiple views of files being edited.

To fill in this sketchy outline, we need to be more precise about the properties of `W`, `K`, and `M`.

Output

Two problems must be addressed by the output mechanism: how a client can draw in its window and how multiple clients can share the screen harmoniously. The choice of output model — bitmaps, PostScript, display lists — is unimportant to the structure of the client, since any model can be implemented by a synchronous, procedural interface, a programming library. Replacing the library will not greatly affect the structure of the client or its environment. The system described here is based on bitmap graphics because that is perhaps the simplest model.

Bitmap graphics has been described before [Guib 82, PLR 85]. Briefly, a two-dimensional portion of memory, possibly but not necessarily visible on the display, is described by a data structure called a `Bitmap`. The data structure may be passed to several graphics operators, of which the best-known is the rectangular operator `bitblt` or `rasterop`, to effect changes to the memory and therefore to the display. Our first guess, then, will make `W` a `Bitmap`. But that ignores the problem of multiple clients with

windows on the same screen.

The concept of 'layers' generalizes bitmap graphics so it applies to overlapping bitmap windows sharing a physical display by storing in the window system complete backup bitmaps for obscured portions of windows [Pike 83a]. By extending the `Bitmap` type to encompass the properties of layers, the standard operators such as `bitblt` can be applied to partially or wholly obscured windows on the hardware display. Clients of a window system may remain unaware of each other, free to draw in their respective windows regardless of overlap and oblivious to changes in the visibility of their windows.

Other systems typically send their clients 'expose' events when the visibility of their windows changes. Such events require all affected clients to run when the display is rearranged, which can cause considerable paging overhead and delay. A rationale for expose events is that some such mechanism is necessary when a user asks a client to change the size of its window, since the original contents of the window will be lost anyway. Unfortunately, flipping between windows is much more common than changing their sizes. Reducing the common case to the hard case therefore obviates an important optimization: a single bitblt executed by the window system can restore a window much faster than can paging in and executing client code. When windows take longer to repair themselves, the entire user interface becomes less dynamic and less comfortable [Pike 88]. Layers permit very responsive interfaces.

On the other hand, the layers model offers no guidance on how to implement resizing, which is a central issue in the design of a window system. It does not however prevent resizing, and there is a clean solution to the problem, which is explained below, in the section on 'Control'.

Layers have other difficulties. Especially on displays with many bits per pixel, layers are expensive in memory, because they maintain off-screen backup memory for invisible portions of windows. But display memory has become much cheaper, and in systems such as the one under discussion, the machine with the display has no other major use for its memory. It is there to maintain the display, to run the window system.

A more telling criticism is that layers require some shared memory, atomicity, and synchronization. The window system must maintain a central data structure describing the configuration of the display, and the clients must not be writing to their windows when that data structure is being modified. That problem is easily overcome, however, using techniques standard in operating systems. For example, `bitblt` could be made a system call, at least when it is operating on a window. In our case, the structure of the interpreter solves the synchronization problem implicitly.

In summary, although layers have limitations, they have important advantages and are used in this system. Output is handled by passing the client processes a variable, `W`, of type `Bitmap` (extended to be applicable to overlapping windows) that the client may use to access its window using whatever graphics package is available. There are no expose events, and resizing is handled by special techniques.

Input

It remains to decide on the types of the variables `K` and `M` that provide the client with access to the multiplexed keyboard and mouse. The type of `K` is easily determined: it can be a synchronous channel that yields integral values identifying the key that has been pressed. It is analogous to a file descriptor on the UNIX system, from which may be read successive input characters. Since `K` will provide only keyboard data, no further specification is necessary; we need not distinguish keyboard data from mouse data, as the mouse information is available only through `M`. `K` does not provide events, it just delivers characters, synchronously, as they become available and are requested. If it were desired to track up and down transitions of the keys, the transitions could still be represented easily as integers. `K` is declared in Newsqueak as

```
K: chan of int;
```

that is, as a channel of integers.

The behavior of the mouse is harder to model. It has three buttons that go up and down and two dimensions of translational motion. The usual solution is to represent the mouse's behavior by a series of events: button down, button up, motion, motion while button down, etc. It is simpler instead to track the mouse by a series of synchronous messages reporting the entire state of the mouse.

The state of the mouse is represented by a data structure:

```
type Mouse: struct of{
     buttons: int;
     position: Point;
}.
```

The variable `buttons` has a bit set for each button that is depressed, and the position is held in a `Point`, a type that is already part of the graphics library:

```
type Point: struct of{
     x,y: int;
}.
```

The actions of the mouse are reported on the channel `M`, defined as

```
M: chan of Mouse.
```

Each time the state of the mouse changes, the current state is made available on the channel `M`. If the mouse is idle, reads from `M` will block. The semantics of communication in Newsqueak implies that mouse-ahead, if desired, must be provided explicitly. (The properties of channels are discussed in the next section.)

A programmer accustomed to an event-driven mouse interface might argue that the synchronous way of handling the mouse is awkward. If the program is waiting for some genuine event such as a button transition, then decoding the complete state of the mouse to look for the transition might seem harder than just waiting until the specified event happens. But before making that decision, we should look more closely at how the mouse is programmed. For simple cases such as waiting for a single button event, it makes no practical difference; the loop

```
do
     e=getevent();
while(e.type!=LEFTBUTTONDOWN)
```

is equivalent to

```
do
     m=readmouse(M);
while(m.buttons&LEFTBUTTON).
```

Imagine, though, that a more complicated condition is to be met, such as the left button being held down while the mouse is inside some rectangle. The proposed interface solves the problem well:

```
do
     m=readmouse(M);
while(!(m.buttons&LEFTBUTTON && pointinrect(m.position, r))).
```

The full programming language may be used to specify the condition. The mouse may be programmed as if it were being polled, which is probably the simplest way to write mouse software. An event-based interface would instead have to be programmed to reconstitute the state so the condition may be tested. Why provide a complex interface when the programming language can already handle all the complexity required? Events add unnecessary complexity to the problem of interpreting the mouse. But we can only avoid that complexity when we can write code to read the mouse channel independently of the code that handles the keyboard; otherwise, the event mechanism is the only way to collect both mouse and keyboard actions. To keep those two tasks separate, we need processes.

Channels, concurrency and multiplexing

Here is the declaration of the type of clients of the window system developed so far:

```
type Client: prog(W: Bitmap, K: chan of int, M: chan of Mouse).
```

Any program of type `Client`, including the window system itself, can be run in a window.

A client program is started by the window system by creating a window and channels for the keyboard and mouse and then by calling the client program as a separate process. In Newsqueak this is written

```
begin client(newW, newK, newM).
```

The window system records the values of the new channels and uses them to send keyboard and mouse data

to the client. The clients execute independently and concurrently and are likely themselves composed of concurrent subprocesses. The clients are true processes, not coroutines; their execution is finely interleaved, and not just switched at I/O time as in mpx or NeWS [Pike 83, Sun 87]. No client can completely dominate the system, even if it is in a tight loop without I/O. The user interface of the window system or its clients does not block because a client is busy.

The channels that carry messages between processes in Newsqueak, and hence within the window system, are synchronous, bufferless channels as in Communicating Sequential Processes (CSP) or Squeak, but carry objects of a specific type, not just integers [Hoare 78, Card 85]. To receive a message on a channel, say `M`, a process executes

```
mrcv = <-M
```

and blocks until a different process executes

```
M<- = msend
```

for the same channel. When both a sender and a receiver are ready, the value is transferred between the processes (here assigning the value of `msend` to `mrcv`), which then resume execution. Except for the syntax, this sequence is exactly as in CSP.

The client must obey a simple protocol to function properly in the system. Because all communication is synchronous, the client must always be ready to receive keyboard and mouse data. If the client misbehaves, of course, deadlock may result. That possibility is covered, below, in its own section.

The clients are typically written as concurrent processes, one reading the mouse, another the keyboard, and others managing the display. These various processes communicate using internal channels. A complete client that connects an operating system's command interpreter to a window, including all processing of keyboard input such as echoing, typing correction, and so on, takes about 100 lines of Newsqueak, distributed across three processes (keyboard, mouse, and display).

The window system itself is also written in Newsqueak, unlike in other language-based systems such as NeWS. The window system is little more than a multiplexer that creates windows and runs clients in them. Its only user interface is that to create, delete, select, and rearrange windows. The main program is a single process that accepts the usual set of parameters `W`, `K`, and `M`, and uses the mouse to select which window receives keyboard and mouse data. When buttons are pressed with the mouse not above any client, the window system activates its own functions. Otherwise it passes keyboard and mouse data to the appropriate client.

The structure of the multiplexing subroutine is straightforward. It maintains an array of data structures describing the environments of its children:

```
type Env:struct of{     # encapsulated world of a window program
    W:  Bitmap;         # screen/window
    M:  chan of Mouse;  # mouse
    K:  chan of int;    # keyboard
};
env: array of Env;
```

The subroutine is a loop that uses Newsqueak's selection control structure, much like the selection operators in CSP and Squeak, to wait for I/O. When keyboard or mouse information is sent, the system decides which client should receive the data, passes the data on, and waits again.

This sounds very much like the usual event-based loop of most window systems and their applications. The main difference is that none of the software needs to be written as state machines. The problems have been decoupled, and the individual components — window multiplexer, clients, and mouse and keyboard handlers for the individual clients — can execute concurrently, independently, and without explicit state. Simpler software results. The multiplexing subroutine, the heart of the window system, is about 60 lines long. Another 100 lines or so is used for ancillary functions such as interpreting mouse motion to define the location of new windows. Graphics and layering operations are provided atomically by a built-in Newsqueak library implemented in C.† The complete window system, capable of running recursively,

† The layer library comprises 364 lines of C, not counted here.

including deleting and rearranging windows (discussed below) is fewer than 300 lines of Newsqueak. A client that provides a simple terminal interface to a command interpreter adds another 100 lines.

Because the interface to the clients is well-defined, the multiplexer knows nothing about the clients themselves. This allows it to multiplex arbitrary programs that satisfy its protocol, including itself. It may thus be invoked again by a client to do submultiplexing within its window. (A small amount of easily arranged protocol is required to initialize the system with the definitions of the functions to be multiplexed.)

We also need the ability to load new clients while the system is running. The major hurdle is linguistic — it is hard to run arbitrary programs in a statically typed language — but soluble. Loadable clients have little affect on the size or structure of the system, but require some strengthening of Newsqueak's type system, which is outside the scope of this paper.

Control

With the approach taken in this system, new forms of communication between the window system and the client are implemented by adding synchronous protocol. A trivial example is changing the mouse cursor, which could be done by having a channel for passing cursor descriptions to the window system. This channel could carry other information, so for generality let this 'control' channel, `C`, hold character strings (arrays of characters in Newsqueak):

```
type Client: prog(W: Bitmap, K: chan of int,
                  M: chan of Mouse, C: chan of array of char).
```

(It would also be fine to have a channel of type `Cursor` for the special job of changing the cursor.) The definition of type `Client` is getting more involved. We can tidy it up a bit by borrowing the `Env` data structure from the main loop of the multiplexer. And we can prepare for the future by providing a pair of control channels, one each way:

```
type String: array of char;
type Env:struct of{     # encapsulated world of a window program
    W:  Bitmap;         # screen/window
    M:  chan of Mouse;  # mouse
    K:  chan of int;    # keyboard
    CI: chan of String; # control messages in
    CO: chan of String; # control messages out
};
type Client: prog(Env);
```

As a more interesting use of the control channels, consider how a client tells the window system that it is exiting. When a client wants to exit, it shuts down internally and as its last action asks the window system to delete its resources:

```
client: prog(e:Env){
     do
          things();
     while(notdone());
     e.CO<- = "Delete";
};
```

Communication is synchronous, so, at the instant the window system receives the client's ''Delete'' message, it knows the client is gone, and it can shut down connections to the client. A similar argument shows that the system should never block because a client is trying to exit.

What happens when a user of the system wants to delete a window? The usual method is to send an event or, worse, an asynchronous poisonous message (a 'signal' in the UNIX system) to the client, terminating it suddenly. This brutality is avoidable; the system can just notify the client, using the same synchronous methods, that it is being asked to exit. When the client is ready, it can exit by the same method as above. In other words, instead of the client being killed, it can be asked to leave. (Part of the protocol of the client is that it must soon honor the request.) This protocol works well for recursively instantiated windows. When a window system is asked to exit, on its `CI` channel, it turns and asks its clients to exit on their `CI` channels. When they have all gone, it then reports on `CO` that it is done.

The same logic can be applied to the other major control problem, which is how to change the location or size of a window on the screen. Again, the usual solution is to change the client's size and then abruptly to notify it of the change. Instead, as with delete, we can install a protocol so the client may ask the system to change its size for it, then add a message in the other direction so the system can ask a client to request a change. In fact, the window system and its client can exchange windows: a channel of type `Bitmap` (that is, a description of a bitmap, not the actual data) can be used to pass the new window to the client, and to return the old one to the window system when it can be deleted. This protocol has the advantage that the client has, for a moment, the old and new windows, and may therefore copy portions of its old window to the new one.

Here is the final definition of type `Client`.

```
type String: array of char;
type Env:struct of{     # encapsulated world of a window program
    W:  Bitmap;         # screen/window
    M:  chan of Mouse;  # mouse
    K:  chan of int;    # keyboard
    CI: chan of String; # control messages in
    CO: chan of String; # control messages out
    CW: chan of Bitmap; # exchanging old and new windows
};
type Client: prog(Env);
```

It might seem that the delete and resize messages are events after all, that nothing has really changed. To be sure, a programmer is still free to write an event-driven application for this system. But by making a conscious change to a concurrent style, much of the discomfort of event-driven interfaces can be avoided. The underlying structure is different. Delete and resize messages are synchronous; all events associated with the mouse have been eliminated, replaced by synchronous reads; and events reporting changes of visibility of a window are unnecessary. The structure of the system — a set of concurrent processes communicating on synchronous channels — makes the control of multiple complex inputs decomposable into small, easily understood components whose design is chosen by the programmer, not the interface. Even in a system where events are unavoidable, that approach makes them easier to manage.

Deadlock

The price to pay for this style is twofold: the need to write in a concurrent language, using novel techniques; and the possibility of deadlock. The novelty of concurrent programming will wear off with practice, but deadlock is harder to dismiss. Although in an experimental system a deadlock is at worst an annoyance, in a production system it is unforgivable. This is hardly the forum for a long discussion on the subject of deadlock, but there are pragmatic considerations worth mentioning.

A deadlock is really nothing more than a peculiar form of bug, and any method for eliminating bugs will work for deadlocks. The best method is to design the system to be bug-free. For deadlocks in particular, that is a practical suggestion. Since the interface between the window system and the clients is completely specified, it can be proven deadlock-free, either formally or experimentally by programs such as `trace` [Holz 88]. `Trace` has the advantage that it is also capable of simulating the communications of the window system itself, so it is possible to make strong statements about the reliability of the system. That simulation requires considerable work on the part of the programmer, however, and is probably an unreasonable demand to make on all applications programmers. Thus the window system should provide its own level of defense against errant clients. Although this has not been done, it should be possible to provide a second order interface around an undebugged client. That interface would be a correct set of processes that honors the protocol to the window system, and connects the client and the window system as long as the client is well-behaved. If the client errs, however, the interface isolates the client from the window system and then enters some state where the client and its errors may be examined.

Specifications and Streams

Much as file descriptors (rather than explicit file names) and conventions about the format of program output make the toolkit approach to text processing possible on UNIX systems, synchronous channels implementing an interface specification permit a piece-parts approach to the construction of interactive applications.

A left-handed person might want to reverse the interpretation of the buttons on the mouse, so the left hand's index finger accesses the same functions as a right-handed user's index finger. All that is required is to interpose a single process, on the mouse channel connecting to the appropriate application or top-level window system, that reverses the `buttons` field of the `Mouse` structure. The affected application remains unaware of the swap.

The simple terminal-emulating client makes no use of the mouse. Imagine that we wanted to provide a simple history mechanism, so that previously typed lines could be selected by the mouse and sent as if typed again. A pair of communicating processes, watching the keyboard and mouse channels connected to the client, could keep track of typed lines of input and retransmit them on the keyboard channel when selected from a menu triggered by the mouse.

If the client process already made use of the mouse, the history feature could still be provided by triggering it on some gesture not used by the client, or by running the client in a slightly smaller window, leaving a banner along the top of its window, invisible to the client, wherein the history menu may be activated.

A more realistic example is to construct a 'chess terminal': a two-part client that acts as a normal terminal in half the window, but recognizes escape sequences printed to the screen as commands to draw chess pieces on a board represented in the other half. Between the window system and the regular terminal program sit processes that operate the resize and delete protocols; the terminal is unaware that it is in a subwindow. Independent of the window system proper, the external channels that connect from the terminal to the operating system's output are similarly manipulated to catch the escape sequences. The advantage of this structure is that the ordinary terminal part of this program is identically the standard terminal emulator; in most systems, a chess terminal must explicitly provide its own terminal emulator, although it may get help from toolkits.

In the implemented system, `bitblt` is a subroutine. If it were instead a message sent on a channel associated with `env.W`, output messages could be similarly modified, say to provide reverse video or to make slides by copying the output to a file. In the next version of the system, this approach will probably be taken, although it adds some complexity to the specification of the interface.

The window system can be run in a window. As the system was being debugged, the new version was often run in a window of the old version, without the usual need to reboot from scratch for each test run. (The situation is analogous to virtual machine operating systems.)

As mentioned above, perhaps the best example of this technique is to use a fresh instance of the window system itself to manage subwindows for a program such as a text editor. Clients can exploit the multiplexing structure of the system without providing their own multiplexing software.

The overall lesson is that by providing a complete specification of the interface between a window system and its clients, it becomes possible to manipulate and interconnect programs by simulating aspects of that interface. Simulation is just the multiplexing of a single connection, so the step from a window system to window programs assembled from piece parts is a small one.

Status

The programming language Newsqueak has been fully designed and an interpreter for it written [Pike 89]. Using the interpreter, I have written a window system that demonstrates the viability of the synchronous design. The window system is usable as a front end to a UNIX system; the functionality it provides is comparable to the standard X window manager or mpx [Xlib 88, Pike 83]. In its ability to run recursively, it offers a unique property. This is no mere trick; it can be used, for example, to run multiple windows on a single connection to a remote CPU server, although that requires some operating system support beyond the scope of this paper. The window system's main lack is that it must be restarted to link in new clients, but that restriction will pass.

Although the design is easily implemented in Newsqueak — the language was built for the task — it can be put together in more traditional environments. Any system that allows synchronous message passing and multiplexing can be used to construct a synchronous window system. The interprocess communication tools in most UNIX systems, particularly pipes plus the `select` or `poll` system calls, are sufficient to implement this design [UNIX 4BSD, UNIX SYSV]. Although such a system would involve substantially more code than the Newsqueak version, it would still be conceptually simpler than a conventional window system.

Conclusions

Window systems are not inherently complex. They seem complex because we traditionally write them, and their client applications, as single processes controlled by an asynchronous, event-driven interface. We force them into the mold of real-time software. They can be simplified greatly by writing them as multiple processes, each with individual, synchronous communication channels on which to receive data and control information. It is possible to write a complete, useful window system, comparable in basic power to commercial systems, using just a few hundred lines of code in a concurrent language. Even in traditional languages, simplicity can be achieved by replacing event-driven interfaces with synchronous interfaces and some easily-provided multiplexing functions. An interface based on synchronous communication allows a novel style of construction that permits interactive applications to be assembled from piece parts, much as in standard UNIX pipelines.

References

[Card 85] Cardelli, L., and Pike, R., ‘‘Squeak: A Language for Communicating with Mice,’’ *Computer Graphics,* **19**(3), pp. 199-204, 1985

[GKS 84] *Draft Proposed American National Standard Graphics Kernel System, Computer Graphics, Special GKS Issue,* Feb. 1984

[Guib 82] Guibas, L. J., and Stolfi, J., ‘‘A language for bitmap manipulation,’’ *ACM Trans. on Graph.,* **1**(3), pp. 191-214 (1982).

[Hoare 78] Hoare, C.A.R., ‘‘Communicating Sequential Processes,’’ *Comm. ACM* **21**(8), pp. 666-678, 1978.

[Holz 88] Holzmann, G. J., ‘‘An Improved Protocol Reachability Analysis Technique,’’ *Softw. Pract. Exp.,* **18**(2), pp. 137-161, 1988.

[Mana 89] Manasse, M., and Nelson, G., DEC SRC, Private communication.

[Pike 83] Pike, R., ‘‘The Blit: A Multiplexed Graphics Terminal,’’ *AT&T Bell Labs Tech. J.,* **63**(8), part 2, pp. 1607-1631, 1983

[Pike 83a] Pike, R., ‘‘Graphics in overlapping bitmap layers,’’ *ACM Trans. on Graph.,* **2**(2), pp. 135-160, 1983.

[Pike 88] Pike, R., ‘‘Window Systems Should be Transparent,’’ *Computing Systems,* **1**(3), pp. 153-158, 1988.

[Pike 89] Pike, R., ‘‘Newsqueak: A language for communicating with mice,’’ *Computing Science Technical Report 143,* AT&T Bell Laboratories, Murray Hill, New Jersey, 1989

[PLR 85] Pike, R., Locanthi, B., and Reiser, J., ‘‘Hardware/software trade-offs for bitmap graphics on the Blit,’’ *Softw. Pract. Exp.,* **15**(2), pp. 131-152, 1985.

[Rose 88] Rosenthal, D., ‘‘A Simple X11 Client Program,’’ *USENIX Winter Conference Proceedings,* pp. 229-242, Dallas, 1988

[Sun 87] *NeWS 1.1 Manual,* Sun Microsystems Inc., Mountain View, California, 1987

[UNIX 4BSD] *Unix Time-Sharing System Programmer’s Manual, 4.3 Berkeley Software Distribution,* University of California, Berkeley, Calif. 1986.

[UNIX SYSV] *System V Interface Definition, Issue 2, Volume III,* pp. 319-321, AT&T, Summit, New Jersey 1986.

[Xlib 88] Scheifler, R. W., Gettys, J., and Newman, R., *X Window System: C Library and Protocol Reference,* Digital Press, Bedford, Massachusetts, 1988.