This paper was presented at the 1991 CMU Workshop on SML.

eXene

Emden R. Gansner
AT&T Bell Laboratories
erg@ulysses.att.com

John H. Reppy*
AT&T Bell Laboratories
jhr@research.att.com

October 31, 1991

1 Summary

EXene is a multi-threaded **X** window system toolkit that we have been developing on top of **Concurrent ML**[Rep91a, Rep90] (**CML**). This paper describes a snapshot of **eXene**'s development, as presented in two talks at the **ML** workshop at CMU.

2 CML overview

Both the implementation and the user's view of **eXene** rely heavily on the concurrency model provided by **CML**[1]. **CML** is based on the sequential language **SML**[MTH90, MT91] and inherits the following good features of **SML**: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. The sequential performance of **CML** benefits from the quality of the **SML/NJ** compiler. In addition **CML** has the following properties:

- **CML** provides a high-level model of concurrency with dynamic creation of threads and typed channels, and rendezvous-style communication. This distributed-memory model fits well with the mostly applicative style of **SML**.
- **CML** is a *higher-order* concurrent language. Just as **SML** supports functions as first-class values, **CML** supports synchronous operations as first-class values. These values, called *events*, provide the tools for building new synchronization abstractions, which are tailored to the application.

*This work was done while the author was at Cornell University. It was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

[1] Conversely, the development of **CML** was strongly motivated by the desire to be able to support user interface systems comparable to **eXene**.

- **CML** provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- **CML** provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other threads packages.
- **CML** uses pre-emptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Pre-emption insures that a context switch will occur at regular intervals, which allows "off-the-shelf" code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- **CML** is efficient. Thread creation, thread switching and message passing are very efficient (performance numbers are given in [Rep91a]). Experience with **CML** has shown that it is a viable language for implementing usable interactive systems.
- **CML** is portable. It is written in **SML** and runs on essentially every system supported by **SML/NJ** (currently four different architectures and many different operating systems).
- **CML** has a formal semantics. In the tradition of the definition of **SML**[MTH90, MT91], there is a formal definition of the core primitives of **CML** (see [Rep91b] and [Rep92]).

To make this more concrete, Figure 1 gives the signature of some of the **CML** concurrency operations. **CML** programs consist of a collection of *threads*, which communicate via typed *channels*. Both threads and channels are created dynamically, using the functions spawn and channel, respectively. Rather than provide operations for communication, as is done in languages such as

```
type 'a chan
type 'a event

val spawn   : (unit -> unit) -> thread_id
val channel : unit -> '_a chan

val always   : 'a -> 'a event
val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val choose    : 'a event list -> 'a event
val guard     : (unit -> 'a event) -> 'a event
val wrap      : ('a event * ('a -> 'b)) -> 'b event
val wrapAbort : ('a event * (unit -> unit)) -> 'a event

val sync : 'a event -> 'a
val poll : 'a event -> 'a option
```

Figure 1: Basic **CML** Concurrency Operations

CSP[Hoa78], **occam**[Bur88] and **amber**[Car86], **CML** provides first-class values, called events, to represent synchronous operations. For example, the functions receive and transmit build event values to describe channel I/O operations. The function always builds an event value that supplies an infinite stream of its argument. The function sync is used by threads to actually synchronize on the operations described by event values. And the operation poll is a non-blocking form of sync; in a situation in which sync would block, it will return NONE instead of blocking. There are also event combinators to build more complex synchronous operations:

choose**.** This constructs an event value that represents the non-deterministic choice of its arguments (note that this choice is made when sync is applied). A choice may involve multiple communications (both receive and transmit) on the same channel, but a thread cannot communicate with itself.

guard**.** This constructs an event out of an event valued function. When sync is applied, the function is called first, and the result is used for synchronization.

wrap**.** This wraps a function around an event value. If the event is chosen in a synchronization, then the function is applied to the result of the event.

wrapAbort**.** This associates an action to be taken if an event is *not* chosen in a synchronization. A new thread is spawned to execute the action.

The power of this approach is that it allows the user to implement new communication and synchronization abstractions. For example, we have found uses for widely varying abstractions, such as remote procedure call, multicast channels and buffered channels.

3 An eXene overview

The motivation for **eXene** comes from the need for an adequate foundation for building interactive systems. Strong arguments can be made for basing the foundation on a high-level concurrent system[RG86, GR92]. This allows the programmer to avoid a variety of complications in dealing with the user interface, especially concerning such aspects as type safety, extensibility, component reuse and the balance between the user interface and other parts of the program.

EXene is based on the following collection of design points.

- *Concurrency.* Concurrency is necessary to support multiple interface contexts in a clean fashion while avoiding a program architecture biased towards the user interface. From a positive viewpoint, threads provide a useful programming abstraction for structuring software, comparable to functions in their utility. **CML** provides the concurrency model for **eXene**.

- *Applicative.* The complexity of user interface systems magnifies the usual problems with mutable values. **EXene** hides state wherever possible.
- *Widget threads encapsulate state.* Since graphical user interfaces consist largely of side effects, the previous item begs the question of where does the state hide. Following the stylistic lead of **CML**, we use threads and channels to encapsulate state. In **eXene**, as in other systems[Pik89, Haa90], concurrency and delegation replace the object-oriented approach adopted by most standard user interface systems.
- *Separate event streams.* Input naturally divides into three classes: keyboard, mouse and control. **EXene** delivers these events as three separate streams, as this makes it is simpler to handle them in most applications.
- *Hierarchical event distribution.* Instead of a central distribution of events, they should flow down the window hierarchy. This allows the programmer to have more control over how events are processed. Functions can be interposed to create new interactions.
- *Limited scope.* There are certain aspects of the **X** window system that **eXene** does not try to handle. These include support for writing window managers as well as low-level color map hooks to support various animation tricks. By avoiding this small collection of special-purpose functions, **eXene** can be a much cleaner system.

4 The eXene library

The **eXene** library is composed of eight modules. Four (`Geometry, Font, StdCursor, ICCC`) provide various utility services, handling points, rectangles, fonts, cursors and interclient protocols. The `EXeneBase` module provides the fundamental types, such as displays, screens, windows, pixmaps and color maps, as well as functions for making and releasing a connection to an **X** server. The `EXeneWin` module fleshes out the window functionality, supplying functions for the creation, deletion and manipulation of windows. **EXene** provides four types of windows: top-level windows, whose parent is (essentially) the screen; pop-up top-level windows; subwindows, whose parents are other **eXene** windows; and input only subwindows. Graphics operations on drawables[2] are supplied by the `Drawing` module. Unlike the mutable, heavyweight graphics contexts used in **X** to specify drawing characteristics, **eXene** uses immutable, lightweight pens. This helps maintain an applicative style and makes the components more modular by removing the programmer's need to manage graphics contexts as a scarce resource. Finally, the `Interact` module provides the mechanisms for handling events. Components communicate through environments, with output environments providing the parent component's view and input environments providing the child's view. An environment is basically a tuple of events. One event corresponds to keyboard events, such as key press and release; another event provides mouse events, such as button down and up, mouse motion, entering or leaving a window, plus the current mouse state. There are two control events:

[2] Drawables include windows and off-screen pixmaps.

one allows the parent to inform the child that it should redraw part of its display or that its window size has been changed; the other allows the child to request various services, such as changing its window's size, from its parent.

Although providing most of the features offered by the **X** protocol and exposing the underlying graphics model, **eXene** provides a qualitatively different feel to the programmer building a user interface, with many of the rough edges found in standard **X** libraries and toolkits gone. Much of this is due to **eXene**'s reliance on concurrency and the environment connection between components; aspects of this will be discussed more fully in the following section. Some of the differences arise from using **SML** as the base language; features such as garbage collection, datatypes, and the reliance on immutable values assist the programmer significantly. In addition, **eXene** provides a number of small features that ease the programmer's job by providing a higher-level approach than **Xlib**. These features include using lightweight, immutable pens that are not tied to a particular class of drawables; having redraw events return the entire list of damaged regions; queueing draw events until the first expose event is received; making the display and screen arguments implicit for any graphics operation on a drawable; and, providing an efficient mechanism for handling window repairs related to copying areas in a natural, synchronous fashion. This last feature provides a particular good example of the value of **CML** events, and is discussed further in Section 6.

5 The eXene widgets

The base **eXene** library provides sufficient functionality to construct any user interface. However, the architecture of the library does not directly support a general framework in which pieces of the interface can be built by various people at various times and then integrated into a single user interface. For this, we introduce a widget[3] layer on top of the base library. This layer provides the additional protocols necessary for cooperation among widgets, as well as their reuse and extension.

A widget in **eXene** is essentially an instance of the following type:

```
datatype widget_t = Widget of {
    attrs : unit -> window_attr_t list,    (* attributes *)
    bounds_of : unit -> bounds_t,          (* size data *)
    realize : {
        env : in_env_t,
        win : window_t,
        sz  : size_t
      } -> unit
  }
```

The program creates widget values and inserts them into some widget hierarchy, the root of which corresponds to a top-level window. In **eXene**, a parent widget controls the external view

[3]For want of a better term, we borrow the **X** term for a graphical object composed of a window and its interface semantics.

and resources of a child; the child makes requests for services from its parent. For example, the `bounds_of` and `attrs` functions allow the widget to specify how big it would like its window to be and what specific window attributes (e.g., background color, foreground color) it desires. For the sake of efficiency, a programmer can construct a complete widget hierarchy before having any of it appear on the screen. During the process of making a widget hierarchy visible, called *realization*, a parent widget creates a window and an input/output environment pair for each child, and passes this information to the child using the child's `realize` function. The child uses the window for display; the input environment provides its only built-in connection to the rest of the widgets.

The `bounds_t` type mentioned above provides a fairly general mechanism for a widget to specify its geometry requirements.

```
datatype dim_t = DIM of {
    base    : int,
    incr    : int,
    min     : int,
    nat     : int,
    max     : int option
  }

type bounds_t = { x_dim : dim_t, y_dim : dim_t }
```

The fields in a dim˙t value correspond to the following semantics.

- The size in the specified dimension is given by `base + d * incr` for some value of `d` subject to `min <= d <= max`, where `max = NONE` corresponds to no upper bound.
- The preferred or natural value for `d` is given by `nat`.

The use of `bounds_t` does not preclude the use of more general constraint systems.

In addition to specifying how widgets communicate, a widget system should provide mechanisms by which widgets can be tailored. **EXene** currently provides four such mechanisms. The simplest consists of value parameterization, in which the widget is written to adapt to additional specifications supplied later, such as the font to use or a callback function to invoke. Graphical composition is another mechanism. The programmer uses the widget hierarchy to construct a new widget abstraction from the set of available widgets. An example of this would be a labeled slider widget, in which a slider and label widget are combined, with the slider's value configured to affect the value displayed by the label. Many widgets in **eXene** have been written to conform to the model-view-controller architecture, in which the control and view of a widget are separated by a specific protocol. For example, the standard collections of buttons in **eXene** are nothing more than combinations of certain views (textual, arrow, toggle switch, check mark, etc.) with certain control semantics (discrete, continuous, two-state, etc.). Both the views and the controllers are available to the programmer, to be used in whatever combinations seem appropriate. Finally, the widget architecture promotes interposition, in which one widget is wrapped in a function that alters its behavior.

The wrapping function might do nothing more than translate keystrokes, or alter the desired bounds or window attributes. As an example more indicative of the power of this approach, a menu can be attached to a widget by wrapping it with a function that responds to mouse presses on the widget by putting up a pop-up menu.

The current version of **eXene** provides versions of most of the typical widgets found in other toolkits. The simple widgets, i.e., those not having children of their own, include scrollbars, sliders, labels, buttons, lists, canvases, text widgets and menus. For composite widgets, i.e., those that support the layout of children, **eXene** provides frames (to add borders), shapes (to constrain a widget's bounds) and layouts (for maintaining its children in a two-dimensional layout of non-overlapping boxes). Particular to **eXene** is the shell widget, which serves as the root of a widget hierarchy and provides the connection between the **X** notion of windows and events, and those of **eXene**.

6 The internals of the eXene library

The **X** window system is a distributed program with the application clients communicating with the **X** server process. The core **X** protocol consists of 211 different messages, divided into 119 request messages, of which 42 have replies, 33 event messages and 17 error messages. Each request to the server has an implicit sequence number (i.e., the first message sent is number 1, etc.). Messages from the server to the client are tagged with the sequence number of the last request processed by the server; this is used to match replies with requests.

Unlike some non-**C** language bindings for **X**, **eXene** is implemented directly on top of the **X** protocol. The only non-**ML** code involved is the run-time system's support for socket communication. This approach of a complete implementation has the advantage of avoiding the **C** language biases of **Xlib**. Furthermore, it provides a demonstration that **SML** and **CML** can be used to implement low-level systems programs without significant loss of performance.

A connection to an **X** server is called a *display*. In **eXene** a display consists of seven threads; Figure 2 gives the message-passing architecture of these threads. The *input* and *output* threads provide buffering of the communication with the server. The *sequencer* thread generates sequence numbers and matches replies with requests. All error messages are logged with the *error handler*; in addition, errors on requests that expect a reply are forwarded to the requesting thread. The sequencer sends **X** events to the *event buffer*, which decodes and buffers them. The *top-level window registry* is a thread that keeps track of the top-level windows in the application and their descendants. It manages a stream of events for each top-level window in the application. The other two display threads manage global resources: the *keymap server* provides translations from *keycodes* to *keysyms*; the *font server* keeps track of the open fonts used by the application.

A display has one or more *screens*, each of which can support different *visuals* and *depths* (e.g.,

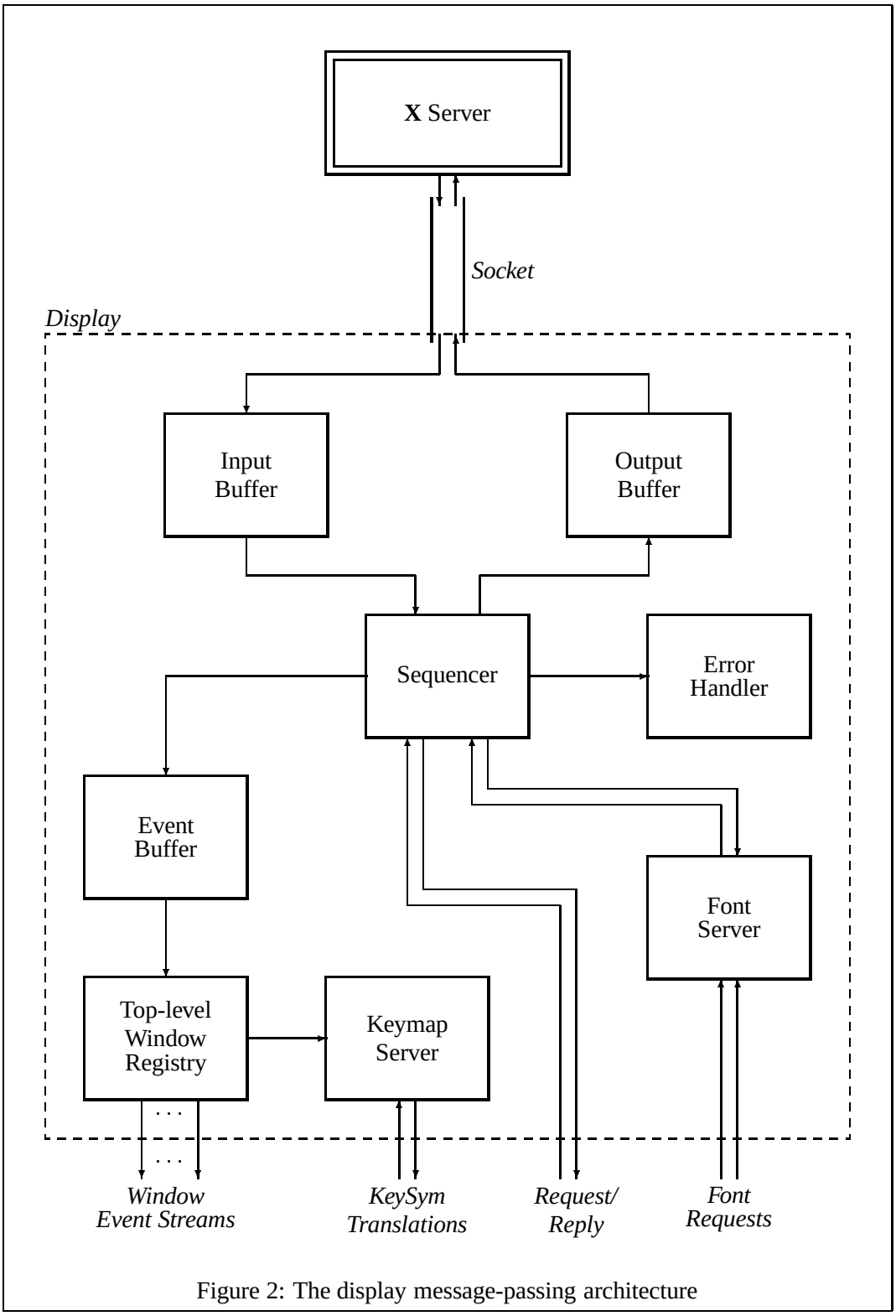


Figure 2: The display message-passing architecture

black and white or 8-bit color). Each visual and depth combination of a screen is supported by two threads; Figure 3 shows the message architecture for these. The *draw master* is a thread that encodes

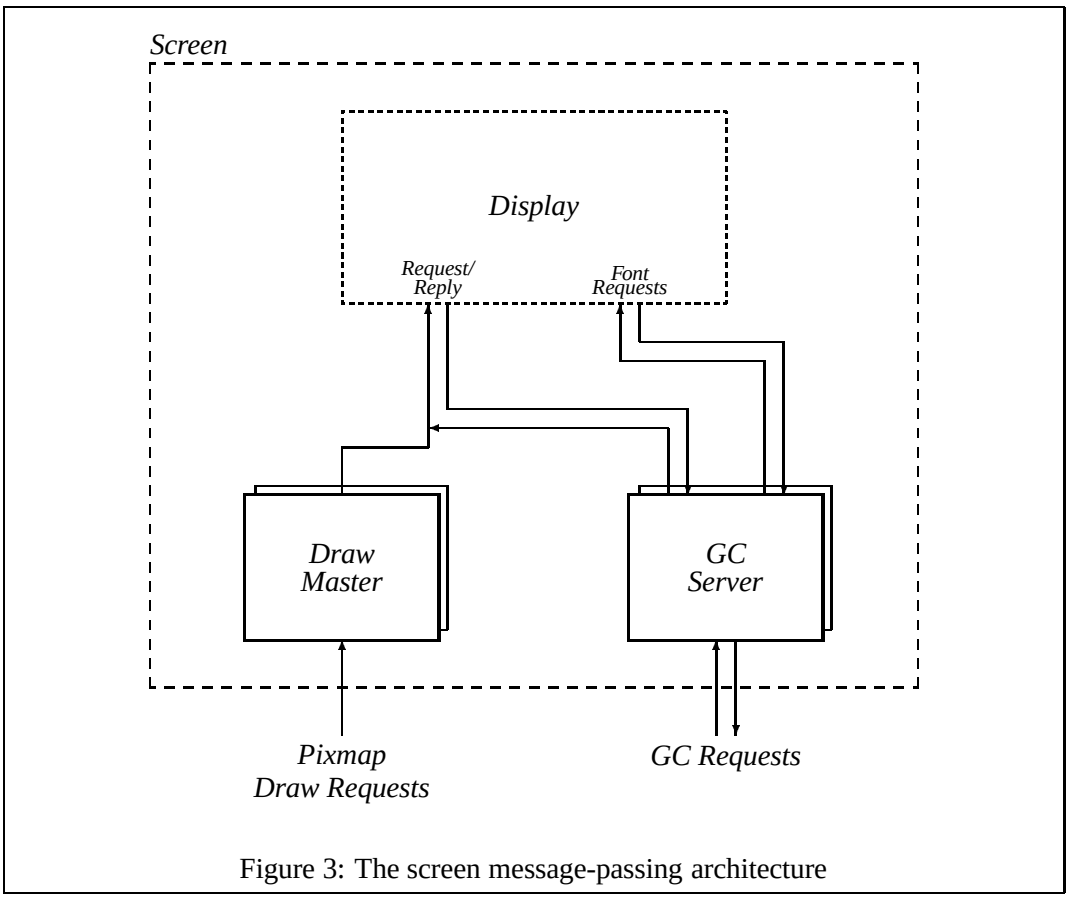


Figure 3: The screen message-passing architecture

and batches drawing requests for a particular visual and depth combination; the draw masters at the screen level are used for operations on *pixmaps* (off screen rectangles of pixels). The *GC server* handles the mapping of **eXene**'s immutable pens to **X**'s mutable graphics contexts[4].

Windows are displayed with a particular visual and depth on a screen. Internally, windows are organized into a tree hierarchy with a top-level window at the root. Figure 4 gives the message-passing architecture for the top-level window threads. As described above, each top-level window in an application has a dedicated stream of **X** events from the display. This stream is monitored by the *top-level window router* thread. This thread provides the transition from the **X** view of events to the **eXene** view (i.e., a window environment). There is a draw master thread for each window tree as well.

[4]It is an unpleasant artifact of **X** that pixmaps and graphics contexts must be associated with a particular screen, visual and depth.

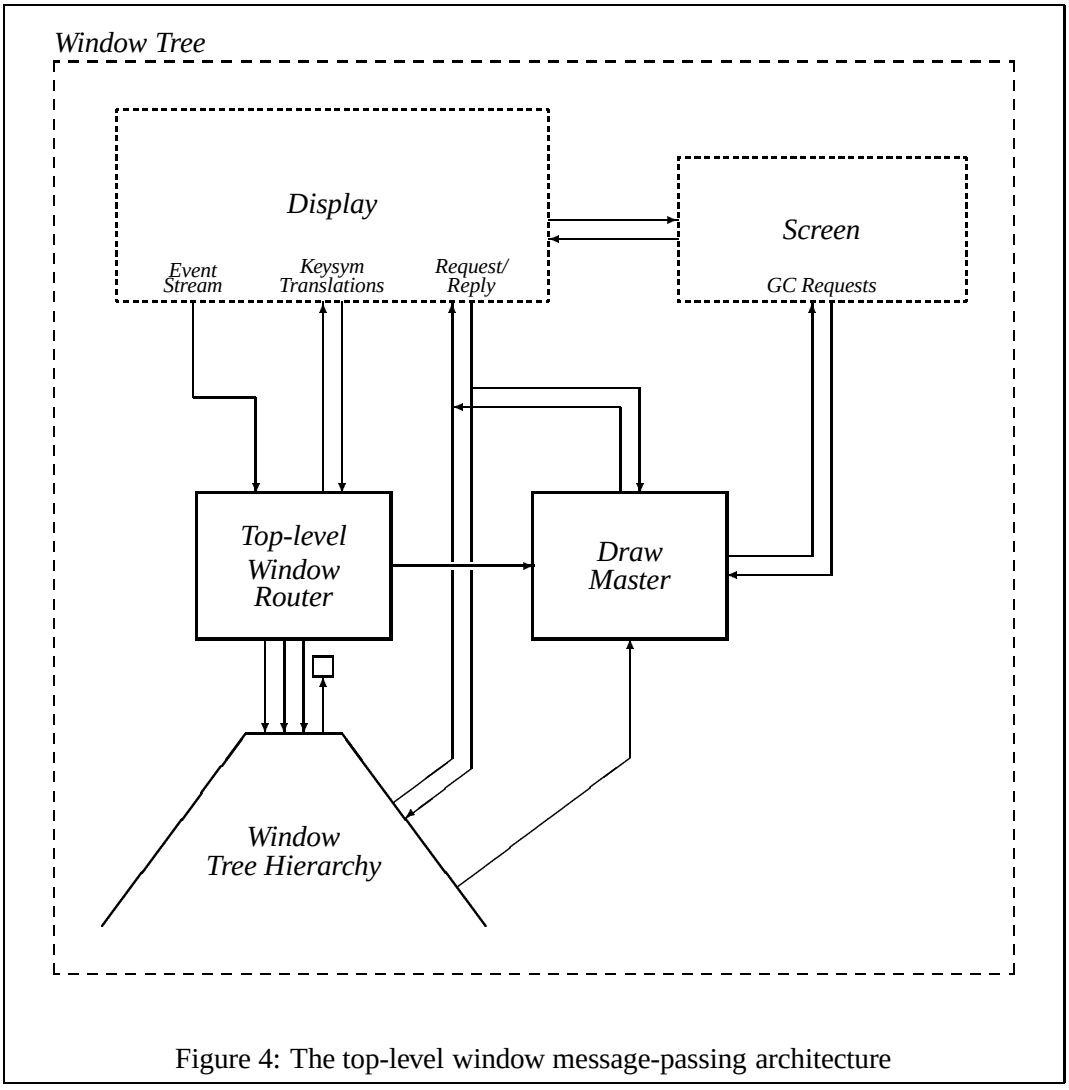


Figure 4: The top-level window message-passing architecture

6.1 Example: `CopyArea`

An interesting example of the use of **CML**'s features in **eXene** is the CopyArea operation, which can be used to copy a rectangle of pixels from one place on the screen to another. A complication arises if a portion of the source rectangle is obscured by another window. For example, Figure 5 shows a use of CopyArea to translate a rectangle on the screen; here the cross-hatched region of the destination corresponds to the obscured region of the source. While some window system maintain

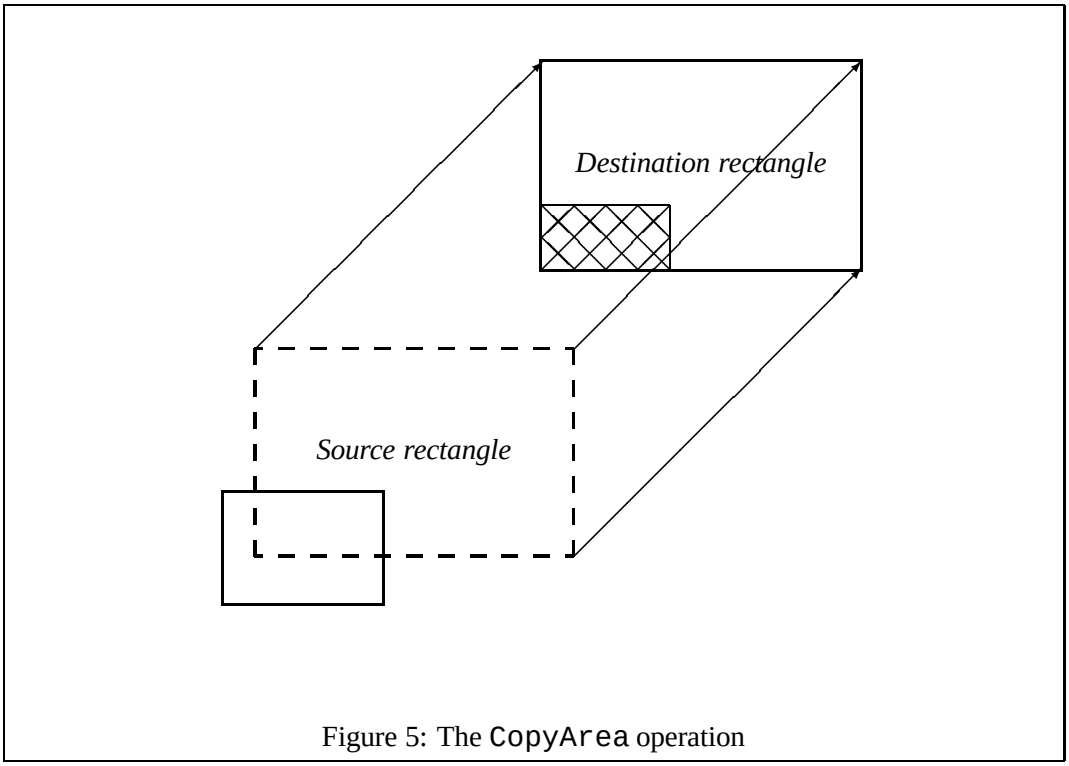
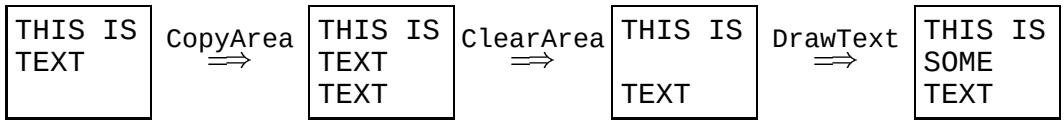


Figure 5: The `CopyArea` operation

a *backing store* (or *virtual bitmap*) to handle these situations, the standard **X** policy is to notify the client that the CopyArea operation was not able to completely fill in the destination[5]. This policy is called *damage control*, since it is up to the client to repair the damage.

A typical use of CopyArea is in inserting a line of text. In this case the client thread might issue the following sequence of operations: a CopyArea to create space for the new text, followed by a ClearArea to erase the old text and lastly a DrawText to insert the new line. The following picture illustrates these steps:

[5] Some **X** servers do support backing store as an option, but applications must be designed to function correctly when it is not available.



It is important that the user of the system see this sequence as a single smooth transition, which has implications for the implementation of operations using CopyArea.

If CopyArea is treated as a normal **X** RPC that returns a list of damaged rectangles, then the user will be subjugated to screen flicker[6]. To understand the reasons for this, examine Figure 6, which shows the timing information for the client doing the text scrolling, the thread handling the buffering of communication with the server (really two threads in **eXene**), and the **X** server. Because the other

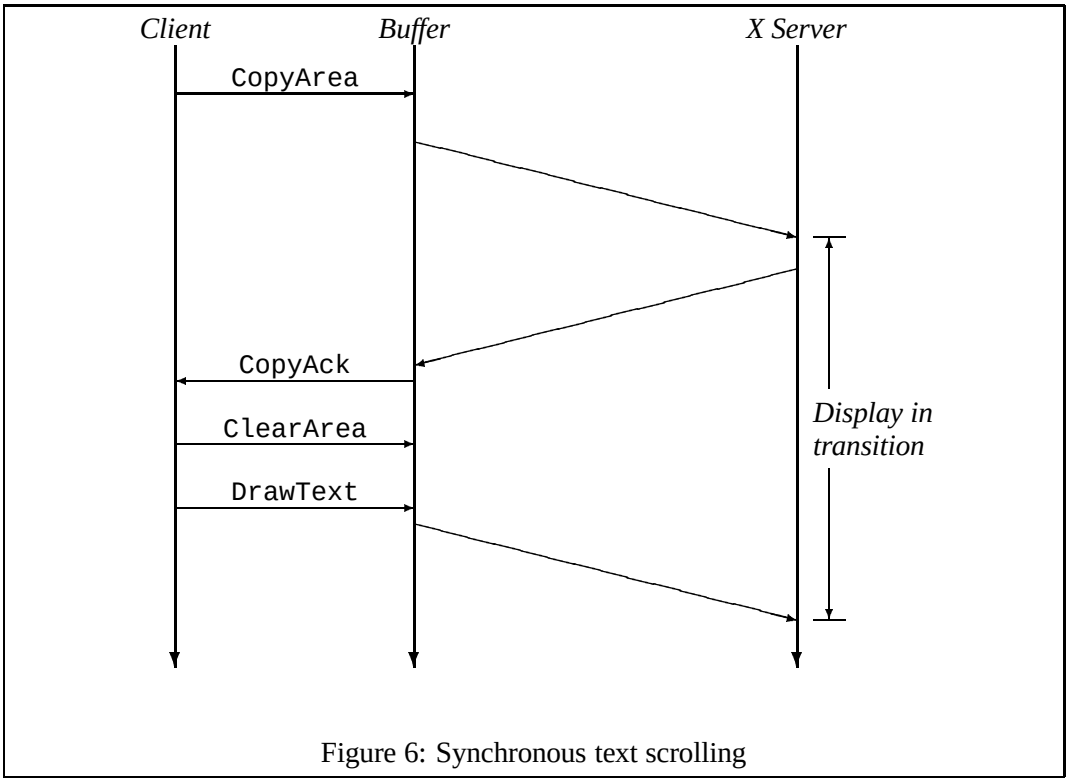


Figure 6: Synchronous text scrolling

drawing operations are postponed until an acknowledgement of the copyArea is received, the period of time the display is in transition can be quite lengthy.

Because of these performance concerns the **X** protocol does not use the standard reply mechanism for CopyArea, but instead uses one of two **X** events, GraphicsExpose and NoExpose, to notify the client of the result[7]. For single-threaded **C** clients (which make up the vast majority of

[6]Practical experience has demonstrated this effect.

[7]Things are a little more complicated, since multiple GraphicsExpose events can be generated for a single Cop-

X clients), this means that the code using the CopyArea operation must also scan the event stream for the acknowledgement. In **eXene**, where we have concurrency and events, we can solve this operation in a much more elegant way. Our solution is to use an *asynchronous RPC*, also known as a *promise* ([LS88]), to support CopyArea calls. **EXene** provides an event-valued function with the type

```
val copyArea : arg-type -> rect_t list event
```

where *arg-type* is the type of the arguments that specify the actual operation. The event that is returned is the promise of the results. This function is easily implemented:

```
fun copyArea arg = let
      val replyCh = channel()
      in
        spawn (fn () => request (COPY_AREA(reply_ch, arg)));
        guard (fn () => (
          case (poll (receive replyCh))
           of (SOME rects) => always rects
            | NONE => (flush(); receive replyCh)
          (* end case *))
      end
```

where request sends the operation to the buffer thread and flush tells the buffer thread to fluch any buffered messages to the server. The guard is optimized to first check if the acknowledgement is already available. The buffer code is more complicated, since it must match the acknowledgements with outstanding CopyArea requests. The advantage of this approach can be seen by comparing its timing diagram, given in Figure 7, with Figure 6.

7 Future work

Although quite usable in its current state, **eXene** is still very much a work in progress. We are already planning various specific changes, some at the implementation level, others providing enhancements to the user's view.

- *X11R5.* The newest release of the X window system includes support for four significant new features: standard, device-independent color models; internationalization; font servers and scalable, machine-independent font representations; and PEX, the X implementation of the PHIGS standard. Some aspects of these features will be incorporated in future versions of **eXene**.
- *Cages.* The Trestle window system[Nel91] uses the notion of cages to specify mouse motion events. Essentially, a cage is a region surrounding the cursor position; the system generates an event when the cursor leaves the cage. This mechanism generalizes the **X** notions of mouse motion (a 1 pixel square cage) and window enter and leave events (a cage corresponding to a window or its screen complement). At present, **eXene** provides no facility by which a

yArea request.

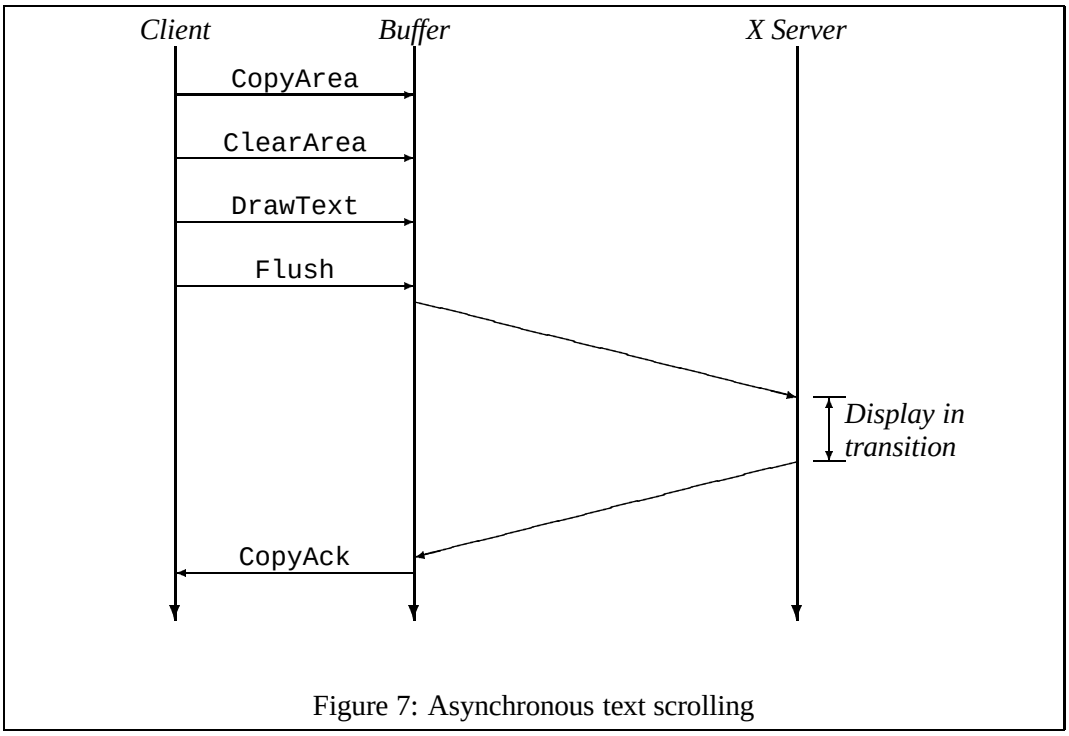


Figure 7: Asynchronous text scrolling

widget can tell the **X** server to ignore unwanted mouse motion events, leading to unnecessary network traffic. It is possible that cages may provide an elegant solution to this problem.

- *Direct event routing.* The hierarchical routing used in **eXene** provides the basis for programmer's ability to wrap an old component in a function providing new behavior. Most of the time, though, events are routed through most paths unchanged. We would like to explore means of maintaining the semantics of hierarchical routing while providing more efficient direct routing when possible.

- *Shape extension.* A fairly standard extension to the **X** protocol provides support for windows of non-rectangular shape. We plan to incorporate this extension into **eXene**.

- *Stub generation.* Much of the code for marshalling and unmarshalling communication with the **X** server is boiler plate code. Because of the many messages and the slight variations between the classes of messages, producing the boiler plate is an error-prone process. We would like to be able to generate this code from tables specifying the **X** protocol.

- *Finalization of system resources.* In the implementation of **eXene**, there is a correspondence between various **eXene** resources, such as fonts and tiles, and their counterparts in the server. Although **eXene** resources can, in general, be automatically reclaimed, this is not possible with those tied to **X** resources, as we must guarantee that the **X** resource is also freed. We plan to attach finalization routines to these resources, which will automatically free the corresponding **X** resources before reclaiming them in **eXene**.

- *More widgets.* There are obvious omissions from the current collection of **eXene** widgets. In particular, we mention a widget for providing panning across a child widget, a composite

widget providing a panes mechanism, and a widget view for radially displayed values, for use in clocks, meters, etc. In general, we prefer to implement a rich set of primitive widgets and allow the programmer to extend them using the mechanisms provided by **eXene**.

- *Different widgets.* Widgets usually correspond to an **X** window. For certain applications, this is too inefficient given the current limitations in **X** and hardware. We hope to explore means by which **eXene** can support more primitive graphical components involving less overhead. This could be viewed as giving widget views a more "first class" status in **eXene**.

Acknowledgments

We wish to thank L. Augustsson, T. Breuel, H. Lin and T. Yan for testing the initial versions of **eXene**, pointing out bugs and suggesting various useful changes.

References

[Bur88] Burns, A. *Programming in occam 2*. Addison-Wesley, Reading, Mass., 1988.

[Car86] Cardelli, L. Amber. In *Combinators and Functional Programming Languages*, vol. 242 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1986, pp. 21–47.

[GR92] Gansner, E. R. and J. H. Reppy. A foundation for user interface construction. In B. A. Myers (ed.), *Languages for Developing User Interfaces*, pp. 239–260. Jones & Bartlett, Boston, Mass., 1992.

[Haa90] Haahr, D. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, June 1990, pp. 289–297.

[Hoa78] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, **21**(8), August 1978, pp. 666–677.

[LS88] Liskov, B. and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 260–267.

[MT91] Milner, R. and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Mass, 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Nel91] Nelson, G. (ed.). *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Pik89] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[Rep90] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual*. Department of Computer Science, Cornell University, Ithaca, N.Y., November 1990. (Last revised October 1991).

[Rep91a] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep91b] Reppy, J. H. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Department of Computer Science, Cornell University, August 1991.

[Rep92] Reppy, J. H. *Higher-order concurrency*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[RG86] Reppy, J. H. and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218–227.