High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System

David A. Nichols, Pavel Curtis,
Michael Dixon, and John Lamping
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304
+1 415 812 4452
{nichols,pavel,mdixon,lamping}@parc.xerox.com

ABSTRACT

Jupiter is a multi-user, multimedia virtual world intended to support long-term remote collaboration. In particular, it supports shared documents, shared tools, and, optionally, live audio/video communication. Users who program can, with only moderate effort, create new kinds of shared tools using a high-level windowing toolkit; the toolkit provides transparent support for fully-shared widgets by default. This paper describes the low-level communications facilities used by the implementation of the toolkit to enable that support.

The state of the Jupiter virtual world, including application code written by users, is stored and (for code) executed in a central server shared by all of the users. This architecture, along with our desire to support multiple client platforms and high-latency networks, led us to a design in which the server and clients communicate in terms of high-level widgets and user events.

As in other groupware toolkits, we need a concurrency-control algorithm to maintain common values for all instances of the shared widgets. Our algorithm is derived from a fully distributed, optimistic algorithm developed by Ellis and Gibbs [12]. Jupiter's centralized architecture allows us to substantially simplify their algorithm. This combination of a centralized architecture and optimistic concurrency control gives us both easy serializability of concurrent update streams and fast response to user actions.

The algorithm relies on operation transformations to fix up conflicting messages. The best transformations are not always obvious, though, and several conflicting concerns are involved in choosing them. We present our experience with choosing transformations for our widget set, which includes a text editor, a graphical drawing widget, and a number of simpler widgets such as buttons and sliders.

KEYWORDS

UIMS, window toolkits, CSCW, groupware toolkits, optimistic concurrency control.

1 INTRODUCTION

Jupiter supports long-term collaboration by providing a shared, persistent virtual world composed of *network places* [7]. By mirroring some of the ways people use physical places, network places provide a context in which

- users can work individually or with other users, by moving between (virtual) places,
- users can organize the materials they are working on and the resources they are using by distributing them among the places, and
- other users, materials, and resources can be brought into an activity at any time, immediately gaining access to the full context of the activity.

In addition, the Jupiter world is intended to be highly customizable by its users, so that they can adapt it to their needs as readily as people adapt physical spaces. This includes facilities for creating new places, connecting existing places, moving objects among places, creating new instances of generic objects, modifying the behavior of existing objects, and creating completely new types of generic objects. These last two capabilities involve writing programs in Jupiter's internal interpreted programming language.

One of the key characteristics of Jupiter is that all objects there are *shared* and *persistent*. This applies to the places themselves, to the documents or other materials, and to the tools. For example, the Whiteboard is a useful generic object that provides a drawing surface with simple sketching tools. As users come and go from the place containing the whiteboard they may open it, see its contents, and edit them; any number of users may be simultaneously viewing and editing it. Even if all the users leave the place, log out of

Jupiter, and come back later, the whiteboard will retain the drawing that was left on it.

Jupiter's central server stores the state of virtual objects and executes all of their associated program code. In contrast, the client programs run by users on their local workstations remain largely ignorant of this virtual-world model. Clients simply manage their local input/output hardware of behalf of the server and the user. In particular, clients create and modify windows according to specifications received from the server, report user input events to the server, and (when available) capture, transmit, receive, and display audio and video data under user and server control [8].

This paper describes the low-level, client/server communications facilities used by the implementation of the windowing toolkit to support Jupiter's pervasive sharing between users. The Jupiter programmer's view of the toolkit will be described in a separate, forthcoming paper.

Jupiter user interactions are subject to two sources of significant response latency, which occasionally cause delays up to a few seconds. First, the serialization of server actions can introduce contention for execution resources. More generally, though, some users connect to the system through high-latency communications paths, such as low-bandwidth dialup lines and long-haul or congested Internet routes. We developed the design presented here in response to these specific sources of latency, but the solutions described would be applicable to any situation with the potential for unacceptable user-event response times.

The server and client communicate in terms of high-level widgets, such as sliders and text editors[1]. Both client and server keep track of widget state, and communicate high-level state changes, instead of low-level user events and graphics primitives. This high-level protocol means they transmit less information, and less frequently, than if a more traditional networked window system (such as the X window system [25]) were used. Because windows can be shared between several users, we need a concurrency control algorithm to maintain consistency.

The concurrency control algorithms use by groupware systems for sharing can be classified as being pessimistic or optimistic. Pessimistic algorithms require communication with other sites or with a central coordinator before making a change to data. This communication can be made apparent to the user, as with a floor control policy, or left implicit, where the user's program does the communication behind the scenes. Even in the latter case, the user must wait for a round-trip to the other sites before any change is finalized.

1. We use "client" to refer to the Jupiter client, which is the program that runs on the user's machine and displays windows on that machine's display. Our "server" runs application programs, which make requests of the client and respond to user events reported by it. The X window system [25] uses these words in the opposite sense; "clients" are applications, and the "server" is the program that displays the windows.

Optimistic concurrency control, on the other hand, requires no communication before applying changes locally. The party making a change applies it immediately, then informs the other parties of the action. If more than one participant makes a change at the same time, a conflict resolution algorithm creates compensating changes to move everyone to the same final state.

Optimistic algorithms are well-suited for high-latency communications channels since the results of a user's actions may be displayed without waiting for a communications round-trip. For these reasons, we chose an optimistic algorithm for Jupiter. The client always applies user changes (such as moving a slider or typing new text) immediately, without waiting for a server response, thereby providing users with immediate feedback.

Another way of classifying concurrency control algorithms is by whether they use a central coordinator or are fully distributed. Because Jupiter already had a central server maintaining the persistent virtual world, it was natural for us to use a centralized architecture.

The combination of these two choices turned out to simplify our system design considerably. While there are example of both centralized and distributed groupware systems using pessimistic concurrency control, the systems using optimistic algorithms are fully distributed. A distributed, optimistic algorithm must be prepared to handle a change from any participant at any time. Much of the complexity for these algorithms comes from this requirement, and getting all the cases right is very difficult.

Instead, Jupiter uses the optimistic protocol only for the individual server-client links. To the server, each client appears to be operating synchronously with respect to the server's actions. The server can thus use a simple change propagation algorithm to keep all the clients updated and in sync.

In the next section, we review related work. Sections 3 and 4 describe the toolkit's design in more detail. In Sections 5 and 6, we describe the two-way optimistic algorithm, which draws from previous work, then describe how it is used to achieve n-way consistency. Section 7 discusses issues related to choosing message transformations, a necessary part of the optimistic algorithm we use.

2 RELATED WORK

The related work falls into two main categories, groupware systems with their approaches to concurrency control, and window systems that cope with low-bandwidth, high-latency communication links.

A number of groupware systems provide either toolkits for building shared applications, or specific shared applications such as text or drawing editors. These include CoEx [20], DistEdit [17], GroupDesign [16], GroupKit [24], the Grove text editor [12], LIZA [14], MMConf [5], Rendezvous [21], Suite [11], and Visual Obliq [3]. LIZA, Rendezvous, Suite and Visual Obliq use a centralized coordinator, while CoEx, DistEdit, GroupDesign, GroupKit, the Grove text editor, and MMConf are distributed.

Of these, GroupDesign and Grove are the only systems to use optimistic concurrency control, and they have decentralized architectures. As discussed above, these algorithms are made more complex by having to support full *n*-way replication. The Jupiter two-way algorithm is derived from the dOPT algorithm used by Grove. This paper extends their work by showing a simplified algorithm that takes advantage of the centralized architecture, and discusses the pragmatics of designing the message transformations required by the algorithm.

Jupiter is similar to Visual Obliq in a different way. Both systems use techniques from the FormsVBT system for Modula-3 [4] and provide tools for rapid prototyping of shared applications. Visual Obliq requires that the application explicitly manage sharing, while Jupiter provides shared widgets. Also, Visual Obliq is not optimized for low-bandwidth or high-latency communications.

Another group of systems deal with low-bandwidth communications channels in single-user window systems. These systems use two basic approaches to dealing with low-bandwidth: compression and code-shipping.

Compression systems include Xremote, Low-bandwidth X (LBX) [13], Higher bandwidth X (HBX) [9, 10], and various commercial programs for providing remote access to Microsoft Windows connections, such as CarbonCopy. They try to reduce the bandwidth used while keeping the same semantics for the network protocol. Although remarkable compression ratios can be achieved (HBX gets a 20:1 compression ratio in some cases), they do not eliminate the round trips caused by user interactions such as keystrokes and mouse clicks.

Jupiter eliminates many round-trips by only sending messages for larger-scale widget value updates, not for individual low-level keyboard and mouse events. However, it requires a trade-off from the application designer: Jupiter applications must use the Jupiter toolkit, and are thus limited to the set of widgets it provides.

The other approach to coping with slow networks is to split the application into two parts, and send the code for one part to the user's machine where it can have fast access to the display and input devices. This is done by Sun Microsystems' NeWS [15] and HotJava [26] systems, and by Bell Lab's Blit terminals [22]. We call this *code-shipping*.

The code-shipping approach is in principle very powerful, since it allows the application to customize the communications protocol to its particular needs. An application can send most of its user interface to the other side, and only communicate high-level events, such as "please run this database transaction with these inputs." The Sam text editor [23] for the Blit is an example of such an application.

```
(VBox
  (HBox
    (Button %help (Text "Help"))
    (Fill)
    (Button %quit (Text (FGColor red) "Dismiss")))
  (Bar)
  (TextEdit %contents (BGColor white)))
```

Figure 1: An example form for Jupiter. Figure 2 shows the resulting window.

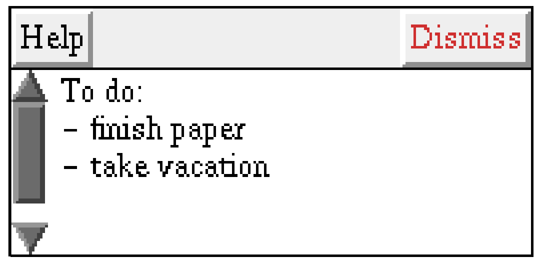

Figure 2: A window created by the form in Figure 1.

However, code-shipping forces the application designer to write a distributed application. The program must be split and a network protocol invented for it. While this is straightforward in some cases, the problems of distributed systems, such as maintenance of replicated state, must be addressed by the programmers of these system. In some systems, the code shipped to the user interface engine must be written in a different programming language from the application, further complicating development.

Again, by trading off some generality, Jupiter is able to hide these network problems from the application. The application programmer sees a conventional toolkit interface, and the protocols and remote code are handled by the system.

3 THE APPLICATION INTERFACE

Jupiter is built on top of an unmodified instance of the LambdaMOO server; all of the server-side facilities described here are written in the MOO programming language [6]. The server contains a database of all the information in the Jupiter environment, as well as a MOO interpreter. Windowing applications are written by Jupiter's users and run in this server.

The programming interface presented to an application writer is very similar to that of the FormsVBT system [1, 4]. A window is described with an S-expression that specifies the types of widgets present and a containment hierarchy for laying them out. For example, Figure 1 shows the description of a window we use for editable documents in Jupiter, and Figure 2 shows how that window appears to the user.

This window consists of a vertical stack of things (VBox), the first of which is a horizontal row (HBox) consisting of a push-button named "help" containing the text "Help", some filler that serves to place the two buttons at either end of the row, and another button named "quit" and displaying the

Layout widgets	
HBox, VBox	horizontal or vertical composition
Fill, Glue, Bar	spacing between widgets
Rim, Border	space around a widget subtree
Leaf widgets	
Numeric	a slider
StrokeEdit	graphical display and interaction
Text	output only
TextEdit	full text editor
TextList	list of text items to choose from
TypeIn	single-line input
Typescript	append-only with user input
VideoPane	displays a digital video stream
Filter widgets, which decorate a widget subtree	
AudioHighlight	flashes a border around the subtree when a user speaks
Boolean	shows a visible on/off value
Button	push once or momentary

Table 1: Widget types available in Jupiter

text "Dismiss" drawn in red. Below this row is a thin black line (Bar), and a text editor widget named "contents".

The name on a widget is used by the application to manipulate the widget (e.g., change its value) and by the toolkit to inform the application of any user interactions with the widget. The name is also used to refer to the widget in the communications protocol.

Table 1 shows a list of the widgets supported by Jupiter. These are mostly the conventional set from other windowing toolkits. The TypeIn widget is for single line text entry. The TextEdit widget is for editing larger (plain text) documents. The StrokeEdit widget is a graphical display and interaction widget, similar to EZD [2] or Tk's canvas widget [18]. The VideoPane and AudioHighlight widgets provide support for audio/video communications.

Each widget type exports a set of operations to the application. For example, the TextList widget, which allows a user to select from a list of text items, supports operations for changing the list of items and for setting which item is currently selected. Widgets with small values, such as Booleans and Numerics, support setting that value. Widgets with complex value types, such as the TextEdit and StrokeEdit widgets, supply operations for incremental update.

In addition, each widget type supplies a set of event notifications, most of which are a result of user interactions. Buttons notify the application when they have been activated by the user. Numerics do so when the user chooses a new value for them. The StrokeEdit widget has a number of user interaction modes, allowing the user to click on existing strokes, or add new ones. When any widget event occurs, a predetermined application routine is invoked with information about the event.

Jupiter applications are sharable, with the toolkit automatically maintaining identical widget values for each participating user. Each change made by a user is reported to the application and automatically propagated to the other users.

4 THE CLIENT-SERVER INTERFACE

Jupiter's users run the client on their local workstation. It makes a TCP connection to the server, running on a central machine. The client program is assumed to evolve slowly, with users obtaining a new one primarily when major protocol changes occur. Currently, implementations for the client exist for several Unix platforms, using the Tcl/Tk toolkit [18], and for Microsoft Windows, using the native Windows programming environment.

The client-server protocol for Jupiter largely parallels the programmer's interface; most calls by applications result in messages to the client, and most client messages generate event notifications to the application. The protocol is designed to use one-way messages, not requiring immediate replies, whenever possible.

When a window is created, the server sends the S-expression describing the window to the client, which creates the corresponding window.

After the window is created, updates to widget values can originate from either the client or server. Updates to simple widgets, such as Numerics and Booleans, include the entire widget value. For user-originated changes, low-level mouse and keystroke interactions are filtered out. For example, the Numeric widget does not send intermediate values while the slider is being moved, but waits until the mouse button is released to send the final widget value.

More complex widgets, such as TextEdits and StrokeEdits, use incremental state update messages. For TextEdits, a general "replace this region of text with this value" message suffices. StrokeEdits use a more complex protocol, shown in Table 2.

Both client and server maintain a full copy of each widget's value. The client copy allows user changes to be reflected immediately in the window, before they are processed by the server. The server's copy is used to coordinate updates generated by the various clients sharing the widget. In addition, the server's copy provides quick access for applications; they do not incur round-trip delays for fetching the values of widgets in order to do computations.

5 THE TWO-WAY SYNCHRONIZATION PROTOCOL

As mentioned earlier, optimistic concurrency control allows clients to change widget values without having to wait for a server interaction. If either the client or the server initiates a change to a widget, the change is immediately applied locally and a notification is sent to the other party. When messages cross on the wire, each receiver fixes up the incoming message so that it makes sense relative to the receiver's current state. The algorithm we describe guarantees that these fixups will suffice, that the client and server

Server Operations	
create stroke, delete stroke	change the set of graphic items being displayed
move stroke	move an existing stroke
set stroke attributes	change color, etc. of an existing stroke
set mode	set a user interaction mode, allowing local creation or deletion of strokes
set creation attributes	set attributes for strokes created by the user
Client Operations	
hit down, hit up	report simple mouse hits of existing strokes
sweep down, sweep drag, sweep up	report strokes hit by a sweep operation that potentially passes through several strokes
create stroke, delete stroke	reports user-initiated creations and deletions

Table 2: Operations available on StrokeEdit widgets

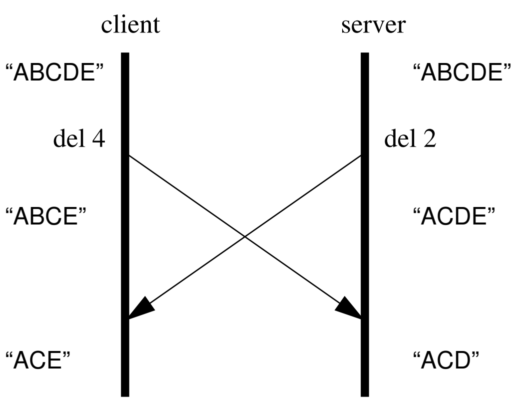

Figure 3: An example of an update conflict. The client has deleted the fourth character, "D", while the server has deleted the second one, "B". Without concurrency control, the client and server wind up with different final values. The fix is to have the server transform the client's message into "del 3" so that both client and server get the same result.

will always agree on the widget value when all messages have been received and processed.

Unlike other groupware systems, Jupiter does not use the synchronization protocol directly between the clients. Instead, each client synchronizes with the server, the server serializes all changes and echoes changes made by one client to all others that are sharing the widget. This lets us achieve n-way synchronization by running independent two-party synchronization protocols on each client-server link.

Figure 3 shows an example of two messages for a single TextEdit widget that cross. If the messages are not transformed on receipt, the client and server wind up with different final values for the widget. Since the client intended to delete the "D" in the original string, its message must be fixed up to read "delete 3" when the server detects the conflict.

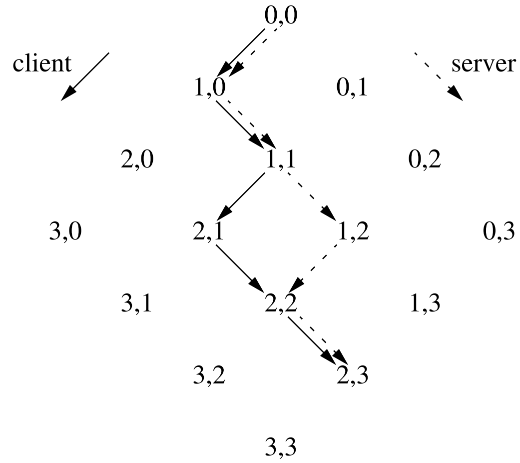

Figure 4: The state space the client and server traverse while processing messages. Each node is labelled with the number of client and server messages processed when in that state. A conflict has occurred starting from the state 1,1.

The general tool for handling conflicting messages is a function, *xform*, that maps a pair of messages to the fixed up versions. We write

xform(c, s) = {c', s'}

where c and s are the original client and server messages. The messages c' and s' must have the property that if the client applies c followed by s', and the server applies s followed by c', then the client and server will wind up in the same final state.

Of course, there are many possible functions that have this property. For example, the function *xform*(c, s) = {delete everything, delete everything} would satisfy our ordering property, but would probably not satisfy many users! In general, the transforms should try to find some "reasonable" way to combine the two operations into a final effect. For our delete example, this is easy to do:

xform(del x, del y) =
{del x-1, del y} if $x > y$
{del x, del y-1} if $x < y$
{no-op, no-op} if $x = y$

That is, we modify the later index in the document to account for the earlier deletion. Other pairs of operations present more difficulties; Section 7 talks about the issues in designing transformations in more detail.

In describing the full protocol, it is helpful to picture the state space that the client and server pass through as they process messages. Figure 4 shows an example. Each state is labelled with the number of messages from the client and server that have been processed to that point. For example,

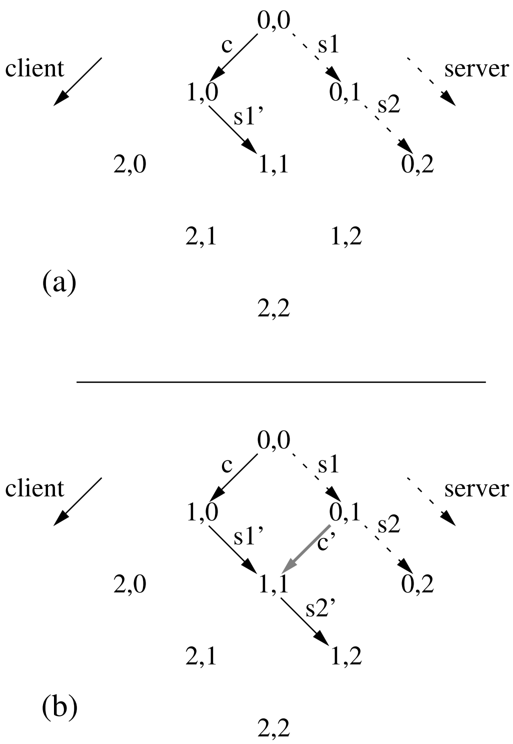

Figure 5: In this example, the server generates two messages, s1 and s2, that conflict with the client message c. In order to process s2, the client must compute c', even though it will never execute it.

if the client is in the state (2,3), it has generated and processed two messages of its own, and has received and processed three from the server.

As each message is processed, the client or server moves down though the state space. If they process messages in the same order (that is, there are no conflicts), then they will take the same path. If there is a conflict, then the paths will diverge, as shown in the diagram. The client and server moved to the state (1,1) together by first processing a client message, and then a server message. At that point, the client and server processed different messages, moving to the states (2,1) and (1,2), respectively. They each received and processed the other's message using the *xform* function to move to state (2,2). Then the server generated another message, sending it and the client to (2,3).

The protocol labels each message with the state the sender was in just before the message was generated. The concurrency-control algorithm uses these labels to detect conflicts, and the *xform* function to resolve them. The algorithm guarantees that, no matter how far the client and server diverge in state space, when they do reach the same state, they will have identical values for all their widgets.

The *xform* function takes a pair of client and server messages that were generated from the same starting state and returns transformed messages that allow the client and server to reach the same final state. As long as the server and

```
int myMsgs = 0;  /* number of messages generated */
int otherMsgs = 0;  /* number of messages received */
queue outgoing = {};

Generate(op) {
   apply op locally;
   send(op, myMsgs, otherMsgs);
   add (op, myMsgs) to outgoing;
   myMsgs = myMsgs + 1;
}

Receive(msg) {
   /* Discard acknowledged messages. */
   for m in (outgoing) {
      if (m.myMsgs < msg.otherMsgs)
         remove m from outgoing
   }
   /* ASSERT msg.myMsgs == otherMsgs. */
   for i in [1..length(outgoing)] {
      /* Transform new message and the ones in
         the queue. */
      {msg, outgoing[i]} = xform(msg, outgoing[i]);
   }
   apply msg.op locally;
   otherMsgs = otherMsgs + 1;
}
```

Figure 6: The algorithm used by client and server to deal with conflicting messages. The pair (myMsgs, otherMsgs) corresponds to the state from Figure 4.

client diverge by only one step, we can use the *xform* function directly. If they diverge further, however, the situation is more complex. Consider Figure5a. In this case, the client has executed *c* and receives the conflicting message *s1* from the server. It uses the *xform* function to compute *s1'* to get to the state (1,1). The server then generates message *s2* from the state (0,1), indicating that it still hasn't processed *c*. What should the client do? It can't use *xform*(*c*, *s2*) because *c* and *s2* were not generated from the same starting state. For example, if *c* is "del 4," *s1* is "del 1," and *s2* is "del 3," then the correct transform for *s2* is "no-op," but *xform*(*c*, *s2*) is "del 3."

The solution is depicted in Figure 5b. When the client computes *s1'*, it must also remember *c'*, the other returned value from *xform*. This represents a hypothetical message that the client could have generated to move from the state (0,1) to (1,1). When *s2* arrives, the client can use *c'* to compute

xform(*c'*, *s2*) = {*c"*, *s2'*}

It executes *s2'* to get to the state (1,2). If the server has processed the client's message, it will be in the state (1,2) as well. If not, its next message will originate from (0,3), so the client saves *c"* just in case.

We are now ready to examine the full algorithm, shown in Figure 6. We describe it from the client's perspective, but the server's actions are identical. The algorithm maintains the invariant shown in Figure 7. The server was last known to be in the state (*x*, *y*). Since then, the client has sent *k* mes-

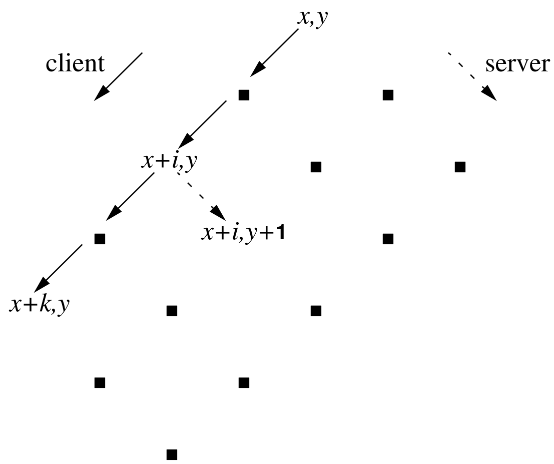

Figure 7: Whenever our algorithm is not processing a message, the situation shown above holds (as seen by the client). The server was last known to be in the state (*x*, *y*). We are in the state (*x+k*, *y*) and have saved all the messages necessary to get from (*x*, *y*) to (*x+k*, *y*). The next server message must originate from some state along this path.

sages, leaving it in the state (*x*+*k*, *y*). These messages are kept in the *outgoing* queue. In the code, *myMsgs* is *x*+*k*, and *otherMsgs* is *y*.

Sending a message while maintaining this invariant is easy: just apply the operation locally to move to (*x*+*k*+1, *y*), transmit it to the server, and then append the message to the outgoing message queue.

For reception, we know that the next incoming server message must originate from one of the states between (*x*, *y*) and (*x*+*k*, *y*) inclusive; that is, the server will have processed some arbitrary number of those *k* client messages. Assume that it comes from state (*x*+*i*, *y*), taking the server to (*x*+*i*, *y*+1). First, discard the saved messages that take us from (*x*, *y*) to (*x*+*i*, *y*), as they are no longer needed. Next, run the incoming message through the transformer with respect to each of the saved messages. The final result will be a message that takes us from (*x*+*k*, *y*) to (*x*+*k*, *y*+1), which we apply locally. While doing this, save the transformed version of each saved message. When we are done, we have a sequence of saved messages that takes us from the last known server state, (*x*+*i*, *y*+1), to our current state, (*x*+*k*, *y*+1). We have thus restored the invariant and are ready for the next incoming message.

Some fine points

In our system, messages must be saved until they are acknowledged by the other party, since they may be needed in order to fix up incoming messages. Normally, these acknowledgments are piggy-backed on traffic going the other way. However, it is possible for the traffic to a window to be one-sided (e.g., for a status display window being periodically updated). Therefore, each side must periodically generate explicit acknowledgments (i.e. no-op messages) to prevent the outgoing queues from growing forever.

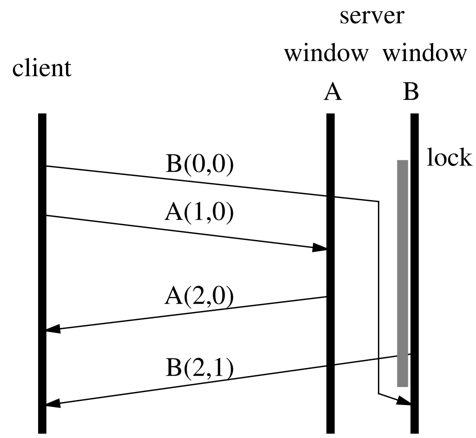

Figure 8: Sequence numbers must be assigned at a grain at least as fine as the locking granularity of the toolkit. Otherwise, an incoming message waiting for a lock on one window might be incorrectly acknowledged by the reply to a message for another window.

An important consideration is the interaction of message numbering with locking. The toolkit requires that applications hold a per-window lock whenever they are examining or changing widget values for that window. If a message arrives from a client while this lock is held, the message cannot be processed until the lock is released.

However, messages must not be acknowledged before they are processed. Figure 8 illustrates this problem. The client sends a message to window B, which has been locked by application code in the server. As a result, processing of that message is deferred until the lock is released. Meanwhile, the client sends a message to the unlocked window A, which generates a reply that acknowledges both of the client's messages. While the window is locked, the application generates a message for B, which also claims to have been generated after both client messages. However, the client's first message actually hasn't been processed yet. The sequence numbers will fool both client and server into assuming the messages don't conflict when in fact they do.

This example shows that locks affecting any message must delay processing of all messages with higher sequence numbers. To avoid unnecessary delays, message counters should be maintained at a granularity at least as fine as the locking granularity the applications use. Our system happens to use window locking, so our counters are maintained on a per-window basis. We could have used per-widget counters, but that would have reduced the opportunity for piggy-backed acknowledgments.

Comparison to dOPT

Our concurrency algorithm is similar to the dOPT algorithm used in Grove. Our *xform* function corresponds to the transformation matrix **T** in their system. In addition to the two operations being transformed, **T** also requires two priorities for tie-breaking. This is because **T** might be applied to operations for any two sites in the system. Because *xform* is

```
/* apply “msg” received from “client” */
apply msg.op to local copy of widget;
for (c in client list for this window) {
   if (c != client)
      Send(c, msg);
}
```

Figure 9: The algorithm used by the server to maintain consistency between clients. This code replaces the line “apply msg.op locally” in Figure 6.

always applied to operations between a client and the server, it can have the tie-breaking rules built-in, simplifying the specification of the transformations.

The dOPT algorithm also has code for saving messages that arrived out of order from a site, then applying them when earlier messages had been processed. Because we do not multicast updates, we can assume a transport layer that delivers messages in order, such as TCP, and omit the reordering code.

Finally, Jupiter’s algorithm fixes a problem with dOPT. When sites diverge by more than one step in the state space, dOPT does not transform saved messages when processing incoming messages (the situation shown in Figure 5). Unfortunately, simply transforming saved messages does not work for the *n*-way case, since the next message can come from a third site that is in an inconvenient message state. Since we handle *n*-way consistency differently, this problem does not arise in our algorithm.

6 GLOBAL CONSISTENCY

The algorithm described in the previous section allows two parties to maintain synchronized widget state. By using a central coordinator, it is easy to extend it to synchronize any number of parties sharing a widget. To do this, the line “apply msg.op locally” from Figure 6 is implemented on the server as shown in Figure 9. As a message arrives from a client, it is sent to all the other clients as well as being applied to the local copy of widget value.

Essentially, the optimistic algorithm hides the asynchrony of the clients from the server. The pair-wise optimistic algorithm makes each client appear to the server to be operating completely synchronously; whenever a client generates a change, it appears that the client has processed all server messages sent so far. If all the clients are synchronous, then the simple algorithm of echoing changes from one to the others works.

It may be easier to think in terms of the two-party guarantees. As long as the server sends to a client every change it applies that the client did not generate, then the two-party algorithm guarantees the client and server will have the same value for the widgets at quiescence. Since each of the clients’ values is equal to the server’s value, they must be equal to each other as well.

7 CHOOSING TRANSFORMATION FUNCTIONS

An important part of using the optimistic protocol is choosing transformations for the pairs of widget operations. While the transformation to use is obvious for some pairs, conflicting requirements in the system make other choices more difficult. This section discusses these concerns, and how we balanced them in Jupiter.

At first, the combinatorics of the problem can seem overwhelming. Jupiter supports 19 client and 24 server messages that directly operate on individual widgets, yielding 456 potential pairs to consider. Fortunately, most of these combinations are uninteresting. Messages can only be in conflict if they are trying to update the same widget, so any messages from different widget types need not be transformed.

The messages for window creation and deletion could cause conflicts if window ids were reused, since one message might refer to an old instance of a window, while another referred to a new instance with the same id. If the server is careful to avoid reusing window ids, we can avoid these problems, and need only consider widget-specific messages.

By considering only conflicts within a widget type, we are left with 65 cases. Of these, 42 are from the StrokeEdit widget, where a number of client messages report user selection of strokes. These messages are similar enough to be considered at once, reducing the number of cases for StrokeEdits to 18 and for all of Jupiter to 41.

To give a flavor for how the transformations work, we show a few examples. Table 3 shows some of the client and server operations for several widgets, and Table 2 from earlier in the paper gives the list for the StrokeEdit widget.

Widget	Client messages	Server messages
Numeric	SetValue	SetValue
TextList	SetValue	SetValue
	Activate	ReplaceItems
TextEdit	Replace	Replace

Table 3: Messages defined for selected Jupiter widgets.

Simple widgets, like Numeric sliders and Booleans, have small values that are always sent in their entirety. For these, we designate one of server or client to be the “winner,” giving

xform(SetValue(v1), SetValue(v2))
= {no-op, SetValue(v2)}

The “losing” message is simply discarded.

For TextLists, transformations for SetValue vs. ReplaceItems and Activate vs. ReplaceItems discard the user action. Each of these client messages is returning the index in the list of text of the currently selected item. However, the application has changed this list, and most likely cannot make sense of this obsolete index. In this case, we believe that the user will usually be able to identify the problem and recover; the application is less likely to be that smart.

The Replace operation for TextEdits deletes a region of text and inserts a string to replace it. For Replace vs. Replace, the transformation produces a final state that (a) has removed all the text requested by either Replace, (b) has inserted the text requested by each, sorted by the starting points of the delete regions. We arbitrarily chose to put server text first if both try to insert at the same spot.

One interesting subcase occurs when one Replace inserts text into the middle of the region of text being deleted by the other. We choose to insert the text anyway, since we want to preserve the user's input if possible. This is true even if the inserted text is coming from the server, since it may be text another user typed that is being echoed to this window.

However, the transformed Replace in this case has to delete two regions, the one before the inserted text, and one after. There is no "delete disjoint regions" operation for TextEdits, so we split the operation into two messages, with the second one implicitly generated from a fractional state number. In general, one has to be careful that a transformation doesn't produce new operations not in the original set defined for a widget. This was the only time we had this problem, and message splitting let us finesse the problem.

The StrokeEdit widget's input modes cause particular problems in choosing transformations. For example, two of the modes a StrokeEdit can be placed in allow users to draw new lines ("line mode") and to click with the mouse to generate reports about which strokes were hit ("hit mode"). If an application switches a StrokeEdit from line mode to hit mode, then it arguably doesn't expect to receive notifications of new lines created by the user. On the other hand, we don't want to discard the user's effort, either. Currently, we keep the strokes, but this decision comes under discussion from time to time.

To summarize, the guidelines we tried to follow were:

- The set of operations should be closed under the transformation rules.
- Try not to discard user input.
- Try to avoid confusing application code.

While we could not always satisfy all these at once, we were generally happy with the compromises we were able to reach.

8 CONCLUSIONS

The Jupiter window toolkit is client-platform independent and provides fast response to low-level user interactions even over low-bandwidth, high-latency communication channels. It achieves these goals through the use of a high level of abstraction on the wire: Jupiter clients and servers communicate solely in terms of high-level widgets and user events.

To solve the distributed-state synchronization problems that this design entails, Jupiter uses an optimistic algorithm with widget-type-specific transformations to recover from server/client conflicts. Jupiter's combination of a centralized architecture and optimistic concurrency control gives us the advantages of both: easy serializability of concurrent update streams and fast interactive response.

Our concurrency-control algorithm is a variant of one developed by Ellis and Gibbs for their Grove text editor. Our contributions include

- significant simplification and improvement due to our added assumption of an ordered, reliable communications channel between exactly two participants,
- a mechanism for doing full *n*-way sharing of widget values using the pair-wise algorithm, and
- a discussion of some of the issues involved in designing the associated transformation functions for pairs of conflicting messages.

9 ACKNOWLEDGMENTS

In addition to the authors, Ron Frederick and Bob Krivacic have helped with the implementation of Jupiter. Early use of the toolkit by Berry Kercheval gave us valuable insight into its design. We are also grateful to Doug Terry and Ron Frederick for reading early drafts of this paper, and to Lisa Alfke for tracking down many papers for us on short notice.

REFERENCES

[1] Gideon Avrahami, Kenneth P. Brooks, Marc H. Brown, "A two-view approach to constructing user interfaces," *Computer Graphics*, 23(3), (July 1989), pp. 137-146

[2] Joel Bartlett, "Don't fidget with widgets, draw!" *Proc. 6th Annual X Technical Conference*, Jan 1992, pp 117-131

[3] Krishna Bharat and Marc H. Brown, "Building Distributed, Multi-User Applications by Direct Manipulation" *Proceedings of the ACM Symposium on User Interface and Software Technology* (UIST '94), 1994, pp 71-81

[4] Marc H. Brown and James R. Meehan, *FormsVBT Reference manual*, available as part of ftp://gatekeeper.dec.com/pub/DEC/Modula-3/contrib/formsvbt.25Mar93.ps.Z

[5] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson, "MMConf: an infrastructure for building shared multimedia applications" *Proc. of the Conference on Computer-Supported Cooperative Work* (CSCW '90), Oct. 1990, pp 329-342

[6] Pavel Curtis, *LambdaMOO Programmer's Manual*, available as ftp://ftp.parc.xerox.com/pub/MOO/ProgrammersManual.ps

[7] Pavel Curtis and David A. Nichols, "MUDs Grow Up: Social Virtual Reality in the Real World," *Proceedings of the 1994 IEEE Computer Conference*, pp. 193--200, January 1994. Also available as ftp://ftp.parc.xerox.com/pub/MOO/papers/MUDsGrowUp.ps.

[8] Pavel Curtis, Michael Dixon, Ron Frederick, and David A. Nichols, “The Jupiter Audio/Video Architecture: Secure Multimedia in Network Places,” to appear in *ACM Multimedia ’95*.

[9] John Danskin, “Previewing PostScript over a telephone in 3 seconds per page,” *Proc. 9th Annual X Technical Conference*, Jan 1995, pp 23-40

[10] John Danskin, “Higher bandwidth X,” *Proceedings ACM Multimedia ’94*, Oct. 1994, pp 89-96

[11] Prasun Dewan and Rajiv Choudhary, “Primitives for programming multi-user interfaces,” *Proceedings of the ACM Symposium on User Interface and Software Technology* (UIST ’91), 1991, pp 69-78

[12] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” *Proc. 1989 ACM SIGMOD International Conference on the Management of Data*, June 1989, pp 399-407

[13] Jim Fulton and Chris Kent Kantarjiev, “A report on Low Bandwidth X (LBX)”, *Proc. 7th Annual X Technical Conference*, Jan. 1993, p 251

[14] S. J. Gibbs, “LIZA: An extensible groupware toolkit”, *CHI ’89 Conference Proceedings*, April 1989, pp 29-35

[15] James Gosling, David S. H. Rosenthal, and Michelle J. Arden, *The NeWS Book*, Springer-Verlag, 1989.

[16] Alain Karsenty and Michel Beaudouin-Lafon, “An algorithm for distributed groupware applications,” *Proc. of the 13th International Conference on Distributed Computing Systems* (ICDCS), May 1993, pp. 195-202

[17] Michael Knister and Atul Prakash, “Issues in the design of a toolkit for supporting multiple group editors,” *Computing Systems* 6(2), (Spring 1993), pp. 135-166

[18] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[19] Keith Packard, “Designing LBX,” *Proc. of the 8th Annual X Technical Conference*, Jan. 1994, pp 121-133

[20] Dorab Patel and Scott D. Kalter, “A UNIX toolkit for distributed synchronous collaborative applications,” *Computing Systems* 6(2), (Spring 1993), pp 105-133

[21] John F. Patterson, Ralph D. Hill, and Steven L. Rohall, “Rendezvous: an architecture for synchronous multi-user applications,” *Proc. of the Conference on Computer-Supported Cooperative Work* (CSCW ’90), Oct. 1990, pp 317-328

[22] Rob Pike, “The Blit: a multiplexed graphics terminal,” *AT&T Bell Labs Tech. Journal*, 63(8) (Oct 1984), pp 1607-1631

[23] Rob Pike, “The text editor Sam,” *Software Practice and Experience* 17(11) (Nov 1987), pp 813-45.

[24] Mark Roseman and Saul Greenberg, “GroupKit: A groupware toolkit for building real-time conferencing applications,” *Proc. of the Conference on Computer-Supported Cooperative Work* (CSCW ’92), Nov. 1992, pp 43-50

[25] Robert. W. Scheifler and James Gettys; with Jim Flowers and David Rosenthal, *X Window system: the complete reference to Xlib, X protocol, ICCCM, XLFD*, Digital Press, 1992

[26] Sun Microsystems, *Hot Java Home Page*, available as http://java.sun.com.