Artefact: A Framework for Low-Overhead Web-Based Collaborative Systems

Jeff Brandenburg, Boyce Byerly, Tom Dobridge,
Jinkun Lin, Dharamaraja Rajan and Timothy Roscoe
Persimmon I.T., Inc.
4813 Emperor Boulevard
Durham, NC 27703 USA
+1-919-941-9339
{jeffb,boyce,dobridge,jinkun,dsrajan,timothy}@persimmon.com

ABSTRACT

The Artefact framework supports collaborative applications using standard browsers, a lightweight general-purpose Java applet, and HTML representations of objects and actions. We present some aspects of Artefact's implementation, including enhancements to HTTP to support synchronous collaboration, the decoupling of input and output in the interaction protocol, and the user agents that bridge the gap between a browser and an application. We describe some of the characteristics that make it easy to create multi-user applications with Artefact, and illustrate this with a simple example application. Finally, we compare Artefact to some existing distributed application platforms.

KEYWORDS: Artefact, multi-user environments, groupware, collaboration, World Wide Web, HTTP, CORBA, update protocols.

INTRODUCTION

The Internet and the World Wide Web have become ubiquitous and familiar platforms for communication, and cooperation. Public access points are becoming common in hotels, libraries and universities, and businesses often have at least low-speed access to Internet services. As access becomes more affordable and widespread, more people are becoming comfortable with Web browsers and the "feel" of Web interaction.

This familiarity and ubiquity makes Web browsers and protocols attractive as a platform for delivering access to many kinds of systems, including collaborative applications. A Web-based system can be used by salespeople or field engineers to interact with one another and their organization from remote locations, perhaps using borrowed equipment and software on other organizations' premises. It can also allow people outside an organization to interact with each other and with organization personnel for applications such as customer support.

Unfortunately, the Web does not make it easy to deploy highly dynamic interfaces, especially synchronous collaborative interfaces. The client-initiated nature of HTTP (HyperText Transfer Protocol) [4] makes it difficult to update information dynamically in response to remote events or the actions of other users. While Java and proprietary clients can solve these problems, they introduce new problems of cross-platform consistency and download delays. In particular, people who want to use an application only occasionally may be reluctant to install a large, specialized client; people who examine an application out of curiosity or mild interest may lose that interest after waiting several minutes for a large applet to download.

The Artefact project represents our effort to address these limitations. Artefact provides an infrastructure for developing distributed object-oriented applications. Developers can use HTML, JavaScript, Java, Tcl, and C++ to define the appearance and behavior of objects. Users can interact synchronously and asynchronously with multi-user applications through standard Web browsers, even over low-speed (14.4kbps or 28.8kbps) connections.

A User's View of the Artefact System

Artefact uses the metaphor of a person moving around in a set of "rooms", referred to as the *application space*, working with application objects such as merchandise, documentation, system resources, or customized application programs. Artefact application objects are the essential building blocks of any running Artefact installation. The Artefact objects within a running system, together with the containment/location relationship that users experience, forms the application space. It is this logical space that Artefact users navigate through. The application space provides a conceptual framework for organising, locating, managing and securing objects, whether they be people, places or things.

The software object that represents a person is called an

"Artefact User Agent" or AUA. The "page" a user views with a browser is actually their User Agent's perspective of a room, including any objects and other users in the room. The browser contacts the AUA through a single URL; as the AUA interacts with different applications, or moves to different rooms, the URL pointed to by the person's browser never changes. A browser connected to an AUA displays a set of frames that comprise the AUA's view of the application space (figure 1).

Figure 1: AUA window

The largest frame displays the room the AUA is currently in. Any objects that the room "contains" are reflected by icons, followed by short textual descriptions, displayed in the room frame. "Exits" to other rooms in the application space are displayed as part of the room. Clicking on one of the exits causes the AUA to move to a new room, updating the room frame to show the new location.

If the user has the appropriate access permissions, objects displayed in the room can be examined more closely, picked up, or moved around. Clicking on the "view" button beside an icon, for example, opens a window displaying the application's user interface. Moving an object allows the user to rearrange the Artefact application space, place an object into their personal space (much like putting an item into a shopping cart), or pass an object to another user. Whenever objects move in the application space, the system updates the appropriate frames of all users' views of the space.

An "inventory" frame for a particular AUA, shown on the upper left of Figure 1, displays the icons of objects which have been picked up by the user. A user might pick up objects representing resources the person needs to manage, personalized documentation, or items being purchased in an electronic commerce application.

The remaining frames of the AUA screen contain text-send and text-receive windows for online chat, and a messages line for the most recent status message. An administrator can customize the size, appearance and layout of all AUA frames as appropriate for specialized applications.

Application Programmer's View

For a site developer, Artefact provides a number of base classes of objects, such as:

- documents: web pages, files
- containers: briefcases, boxes, filing cabinents
- collaborative tools: whiteboards, shared browsers, address books, buddy lists
- autonomous agents: search bots, knowledge bots
- gateways: Internet Relay Chat (IRC) repeaters, SNMP monitors
- scriptable objects: Javascript and Tcl interpreters

Every object supports Access Control Lists to allow fine-grained security controls.

In addition, Artefact provides a complete object-oriented SDK for developing customized application objects. Advanced developers can use C++ and Java to create new object classes; content developers, graphic designers, and casual programmers can use HTML, JavaScript and Tcl to specify object appearance and functionality without in-depth OOP experience. New components can be added to a running system with no interruption of service. All Artefact objects communicate via the CORBA standard, allowing applications to be distributed transparently across multiple servers.

IMPLEMENTATION

Artefact applications are built from objects that communicate among themselves through CORBA interfaces. This makes it easy to distribute application functionality across a number of hosts; in addition, CORBA provides facilities for on-demand server restart and connection re-establishment, which improves robustness in the face of program or communication failures.

Artefact installations can be made completely self-contained and private, with no dependencies on any servers or facilities outside the corporate firewall. Users connect with their AUAs using enhanced HTTP connections, with no CORBA traffic passing enterprise boundaries.

While some Web clients now support direct CORBA communication, the efficiency and portability issues discussed above led us to retain HTTP as our path to the browser. We use the AUA as a bridge, communicating with a browser via an enhanced HTTP protocol and with the rest of an Artefact system via CORBA.

HTTP Enhancements

We chose to use HTTP as our basic protocol for several reasons:

- User expectations: from experience with the World Wide Web, users are familiar with browsers and how they work,

and also how transient network failures can affect responses. While we have changed the interaction model in Artefact from a document-retrieval paradigm to one closer to window system interaction, we have retained some features of the Web that users find reassuring (for example, the use of the "reload" button).

- Security configuration: HTTP is well understood by proxy servers and firewalls, simplifying network management within an organization. Further, inside the AUA it is much easier to validate an HTTP request than (for example) a CORBA/IIOP invocation.
- Installed base: a sophisticated implementation of HTTP already exists in Web browsers.
- Debugging: from our point of view as system developers, the text-based nature of HTTP makes it easy to debug.

The principal drawback to using HTTP in this application is, of course, that it is primarily a client-server, request-response protocol which has no provision for the server to send asynchronous updates to the user's screen. We considered using the "server-push" facilities of some browsers (whereby the server sends a multipart MIME message to the browser, which displays each part in succession as it arrives), but this is somewhat clumsy for our purposes: it requires an open TCP connection for each frame of the display, and is not robust in the event of connections going away. The consequences of server-push for human users are that the browser appears to be loading a document all the time (removing a piece of feedback that we have found useful in practice), and that Artefact would "go dead" if the connection to the browser was interrupted (for instance, with the "stop" button). We felt that both of these compromises were unacceptable.

The solution we adopted involves downloading a small (less than 5K) Java applet to the browser. This applet establishes a single TCP connection to the HTTP port on the server, and thereafter listens for messages from the AUA indicating which frames in the browser need to be updated. The applet then initiates HTTP requests from the browser to update the frames.

This is a lightweight approach with a number of advantages. First, the applet is small, and has no user interface at all, making it highly reliable and portable: we have successfully used Artefact with versions 2.02, 3.0 and 4.0 of Netscape Navigator, and with some versions of Microsoft Internet Explorer 3.0 and 4.0. Second, every message sent by the browser to the AUA is a valid HTTP request. This makes it easy for us to validate the request at the server end, traverse proxy servers, and so forth.

Decoupling input and output

HTTP requests in Artefact are split conceptually into two types: *content requests*, and *input events*. Content requests are generally initiated by the remote control applet, or by users clicking "reload," and return the contents of frames. These requests are idempotent, which increases resilience and preserves a useful property of the WWW: one can always hit reload to update one's screen if something appears to go wrong.

The second type of request, input events, is caused by user activity, such as clicking an active area of a frame, closing a window, etc. These requests do not return any content themselves (they return an HTTP "no content" message), but will generally cause an update notification to be sent to the browser at some later time. In this way, input to application objects (including the user agent itself) and output to the screen from application objects are effectively decoupled.

This decoupling is a big advantage for a developer writing a shared application object, since it separates the business of updating the appearance of the application object from that of sending that appearance to the screen. The decoupling is carried through into the CORBA world: each application object can be queried for its appearance at any time, and propagates events to other interested application objects (including user agents) when its appearance changes. The drawback of this approach is that it does not address the need for instant feedback to the single initiator of each input. In practice this issue is often hidden by the latencies of the Web itself relative to the communication between application objects: by the time the browser has received the "no content" acknowledgement of the input, there is usually a resulting update notification already pending, and we can rely on the "loading document" animation of the browser to provide this feedback.

For each frame of the user's browser, the AUA instantiates an `update protocol' instance which handles HTTP requests for the content of the frame and sends update notifications back to the remote control applet in the browser. This protocol ensures that the frame is not "out of date", and also batches change notifications so that the user viewing the browser is not overwhelmed by a constantly updating frame.

The Artefact User Agent

AUAs manipulate other Artefact objects through CORBA method invocations. These methods allow an AUA to move from location to location, to move other objects, to obtain views of other objects, to send or broadcast messages, and to invoke object-specific actions. Since AUAs are first-class objects, other objects can also obtain a AUA's views or send it messages. An event-distribution protocol notifies objects when a change happens in their location, in their contents, or in an object they are explicitly observing.

To begin using an Artefact application, a user connects to her own AUA. This is usually accomplished through a login process that validates the user's identity; the login facility can also return a cookie that serves as an authentication token for the rest of a session. Once the user is validated, the login process redirects the browser to a URL that encodes the identity of the user's AUA and the address of the server on which that

AUA runs.

When the server receives an HTTP request for this URL, it examines the URL for the *marker*, or unique-within-the-server identifier, of the AUA. It then forwards the request to that AUA's request queue. The AUA replies to this initial request with a frame set; each frame in the set, in turn, loads its contents from another URL referring to the same AUA.

One of these frames loads the remote-control applet. When this applet starts running, it establishes a TCP connection back to the AUA from which it was loaded. When the AUA determines that the content of a frame has changed, it sends an *update request* along this connection; this request contains the name of a frame, and the URL from which it should be loaded. When the remote-control applet receives the request, it uses the Java showDocument() method to make the browser reload the frame.

The user interacts with the Artefact application through standard HTML input elements, such as links and forms. As with the initial frameset, the URLs associated with these elements refer to the AUA and its server. But when the user selects one of these elements – for example, when she follows a link – that request does not return a response, as explained above. Instead, the AUA interprets the request and performs the appropriate CORBA method invocations to carry it out, then reports the results through the update protocol.

This decoupling of input and output reduces the potential for confusion in the face of reloads or repeated clicks. Reloading a frame (or the entire frameset) only refreshes the screen; it does not repeat any commands. Clicking multiple times on a link does submit multiple requests, but by making commands idempotent – for instance, by avoiding "toggle" commands in favor of "on" and "off" commands – we can create interfaces that respond predictably to these multiple commands.

Besides controlling frames in the basic frameset, the remote-control applet can open new windows. The AUA associates "examine" actions with most objects; by selecting an object's "examine" link, a user can open a separate view on that object. Like other frames, the applet updates this view dynamically as the examined object changes.

In busy systems, when an AUA is in a room with many active objects, change events can arrive so quickly that they outrun the update mechanism. For example, if a user is connected over a 28.8Kbps link, and a room's view contains 3 Kbytes of HTML, the view cannot be updated more rapidly than approximately once per second. To avoid communication backlogs, the AUA batches updates. Between the time it sends an update message to the remote-control applet and the time it receives and services the resulting HTTP request, the AUA simply collects changes; only after the request is serviced does the AUA send a new update message. This protocol adapts to varying latency and available bandwidth, so the user always sees a relatively current view – delayed by the network latency between the AUA and the browser, and by the time needed to send the view's contents across the connection, but not by backlogged responses.

EXAMPLE

Objects are created in Artefact by defining their behaviour and appearance in an XML-like format called ADL. Figure 2 shows the complete definition of a very simple TicTacToe game in ADL.

The object is defined at the top level by specifying a name (`TicTacToe`) for the object, together with a *class*, which specifies the underlying implementation to be used. In this case, the class is `TclObj`, which provides an interpreter in the popular embedded language Tcl[16] for controlling how objects behave. The body of the definition contains two main sections: the `BEHAVIOR` and `HTML` elements.

The HTML section defines static aspects of the object's appearance, using a superset of HTML 4.0. The use of HTML in this case allows us to make use of the considerable layout capabilities of modern Web browsers in creating the user interface for objects. More importantly, it provides a minimal learning curve for Web developers new to the Artefact system. New objects can be defined from many preexisting classes (for example, IRC gateways or SNMP monitoring objects) entirely using HTML.

The TicTacToe example shows the use of two elements not present in HTML 4.0: the `SUBST` and `ACTION` tags. The `SUBST` tag is used to substitute dynamic content provided by the implementation of the class. In the case of Tcl objects, the expression to be substituted is simply handed to the interpreter for evaluation, and so the expressions we see in the file are really Tcl command invocations.

The `ACTION` element is the simplest construct available for handling user input in the system. It is analogous to a conventional HTML anchor or link: if a user clicks on the area defined by the tag, the AUA interprets the request and invokes the object's `action()` method, passing as parameters the command string corresponding to the action (in this example the string "reset") and the identity of the user who caused the input. More sophisticated constructs are available for handling other user input features of HTML such as forms.

In contrast to the static declaration of appearance in the `HTML` element, the `BEHAVIOR` element for scriptable objects (like this example) defines how the object behaves in response to the actions of other objects around it (including, but not limited to, human users). Artefact currently supports two classes of fully scriptable objects: TclObj, illustrated here, and a second based on a JavaScript/ECMAscript interpreter.

The `onevent` procedure is called for each event encountered by the object; in this case the only type of events we are interested in are inputs from users of the object. These either cor-

```
<ARTEFACT CLASS="TclObj" NAME="TicTacToe">
<SHORT>TicTacToe board</SHORT>
<BEHAVIOR>
array set board ""
set curmove 0
proc onevent {} {
  global curmove board
  if { [event type] == "action" } {
    if { [event cmd] == "reset" } {
      unset board; array set board ""
      set curmove 0
    } else {
      set board([event cmd]) $curmove
      incr curmove
    }
    refresh
  }
}
proc cell { loc } {
  global curmove board
  if { [info exists board($loc)] } {
    if { [expr $board($loc) % 2] == 1 } { write "O" } else { write "X" }
  } else {
    writeaction "-" $loc
  }
}
proc tomove {} {
  global curmove
  if { [expr $curmove % 2] == 1 } { write "O" } else { write "X" }
}
</BEHAVIOR>
<HTML>
<HEAD><TITLE>TicTacToe</TITLE></HEAD>
<BODY>
 <CENTER>
  <H1>TicTacToe</H1>
  <P><SUBST EXPR="tomove"> to move.</P>
  <P><TABLE BORDER><TR>
    <TD><SUBST EXPR="cell 00"></TD>
    <TD><SUBST EXPR="cell 01"></TD>
    <TD><SUBST EXPR="cell 02"></TD>
   </TR><TR>
    <TD><SUBST EXPR="cell 10"></TD>
    <TD><SUBST EXPR="cell 11"></TD>
    <TD><SUBST EXPR="cell 12"></TD>
   </TR><TR>
    <TD><SUBST EXPR="cell 20"></TD>
    <TD><SUBST EXPR="cell 21"></TD>
    <TD><SUBST EXPR="cell 22"></TD>
   </TR></TABLE>
  <P><ACTION CMD="reset">Reset the board</ACTION>
 </CENTER>
</BODY>
</HTML>
</ARTEFACT>
```

Figure 2: Complete TicTacToe application definition in Tcl and ADL

respond to resetting the board (the “action” described above), or a click on an unoccupied cell in the board.

The other two Tcl procedures shown are called from the HTML section. The first generates each cell of the board, including generating a clickable area for cells which have not been filled in yet. The second simply prints whose move is next. The end result is shown in figure 3.

Note that this fully shared, multiuser TicTacToe game is implemented in 27 lines of Tcl and 23 lines of HTML. Artefact's scripting API is event driven, and hides much of the complexity of the full C++ API (threads, CORBA, persistence, etc.) to provide an easy to learn programming model which is well suited to rapid prototyping.

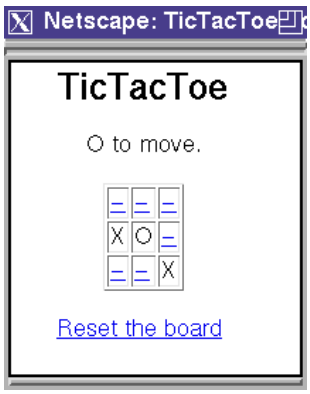

Figure 3: How the TicTacToe application appears

This example was stripped down for inclusion in this paper, but it does illustrate how much can be achieved with very little code. A fuller example might detect wins, and implement a floor control policy assigning players on a first-come-first-served basis, calling them by name: such information is made readily available to script authors. There is clearly room for improving the application cosmetically, for instance using graphics images rather than simply “X”, “O” and “-”: this is simply a matter of enhancing the HTML portion of the object definition.

MORE COMPLEX CLIENTS

As previously discussed, Artefact allows site developers to declare an object using only an ADL file. In the simplest cases, these files do little more specify the object's Artefact class, its name, and HTML description.

Even though objects based on ADL files are sufficient in many collaboration situations, some applications require more fine-grained interaction. To support such applications, Artefact can serve as a “switchboard” for out-of-band communication among specialized applets or clients. As an example, we have built a shared white-board application that allows collaborators to simultaneously draw or type on a shared workspace.

The Whiteboard application differs from other Artefact applications based on a simple ADL definition file, in that much of its user interface is implemented in Java. When a user opens a Whiteboard object, the user's browser loads an applet from the Artefact server. The applet uses a conventional mouse- and keyboard-based interface to draw images or text. The applet also opens a TCP/IP socket back to its corresponding Artefact object. Through this socket, the applet sends drawing commands to the Artefact object which,

in turn, broadcasts the commands to instances of the applet running in other users' browsers. In this way, all applets receive the drawing commands and update their drawing areas. Moreover, the drawing commands are stored persistently by the Whiteboard Artefact object; when other users later open the same Whiteboard, their applets get up-to-date images immediately. Finally, since the Whiteboard object that serves up the applets also serves as a central distribution point for update information, it overcomes the security restrictions which make it difficult for applets to communicate directly with one another.

The Whiteboard application shows that the Artefact framework is not limited to HTTP protocols and HTML documents, but allows a natural mechanism for out-of-band communications using a complex client. Artefact can easily support specialized user interface agents with Java applets or other browser plug-ins. These applications can bring extended capabilities to clients and applications that require them, while still benefitting from Artefact's persistence, access control, and event-handling mechanisms.

RELATED WORK

Previous distributed collaborative systems have taken various approaches in distributing functionality. Systems may replicate state on each client, distribute state across clients, or maintain state in a central server.

DistView [18], COTERIE [10] and DIVA [21] replicate object state on each client. This approach yields good response time, since users interact with local copies of objects. However, replication increases client size and complexity, and can entail significant overhead at startup or installation time as initial state information is propagated into a client.

Distributing system components to clients makes sense when a single client dominates the traffic to any given object. Again, local objects lead to excellent responsiveness, but impose heavier demands on clients. When clients must interact with remote objects, responsiveness suffers. Suite [3] was an early example of this approach; more recently examples include GroupKit [19], CommonPoint [15] and CBE [9].

Centralized systems support applications on a single server, with users accessing applications through relatively lightweight clients. Rendezvous [17], Jupiter [14] and TeamRooms [20] are examples of this approach. While clients are general-purpose within each system, they do not interoperate among different systems. CVW [22] does support a browser-based Java client, but this client is not publically available, and it is unclear how large or complex the client is. (The client runs on Unix and Windows, with a “very limited version” available for the Apple Newton, but not the Macintosh.) Habanero [12] represents a hybrid approach, in which servers may run remotely or on the same machine as a client; again, the Habanero client is not browser-based, and is quite large.

Systems in which clients interact with remote objects must cope with network latency and bandwidth restrictions. Artefact's basic structure is targeted at applications and communication modes in which sub-second latency is not critical. Instead of minimizing update latency, Artefact minimizes network traffic, batching updates to dynamically match available bandwidth and latency. For systems that must support more fine-grained mutual interaction among users, enhanced update protocols [5] can bring update performance close to the theoretical limits imposed by two-way latency.

A number of commercial products now provide collaborative capabilities. Ding [1], ichat [7] and ICQ [11] provide text- based chat and “buddy list” functions; Netopia's Virtual Office [13], TeamWave [23] and eRoom [8] provide virtual office environments, with various combinations of synchronous and asynchronous communication features. For each of these products, users must either install a proprietary client program prior to using the system, or download a large, complex Java applet at the beginning of each session.

CONCLUSIONS

Artefact's lightweight, HTML-based interface supports useful collaborative applications using standard browsers and low-bandwidth connections. We have created testbed applications implementing a virtual office and a problem ticket management system, and we have used these systems without modification with clients running under Solaris, Digital Unix, Windows 3.1, Windows 95, Windows NT 4.0, and Macintosh System 7.5; from desktop and laptop machines, ranging from a 333-MHz DEC Alpha to a 40-MHz 68030; and over modem connections as slow as 14.4 Kbps. While the slowest systems and the slowest links do provide noticeably slower performance, sometimes taking several seconds to perform a single update, they still are responsive enough for text-based conversations and simple, non-graphics-intensive application interaction. Even on these slow systems, it takes less than twenty seconds to start up the Artefact frameset and update applet.

We have used the Artefact testbed applications to let users on both sides of the Atlantic interact with application objects. While the increased network latency does slow down responses, the subjective feel of the application remains quite good. The basic structure of Web browsers and protocols is designed to cope with potentially high latency and low bandwidth, and users with Web experience are familiar with the variations in performance that result. We have not done extensive quantitative tests on Artefact's performance as latency and bandwidth vary, but since Artefact uses the same protocols and presentation languages (HTTP and HTML) as the rest of the Web, we expect that its performance will scale in the same way.

Where more bandwidth and more powerful client machines are available, Artefact can use out-of-band techniques to support richer forms of interaction. We have already implemented a Java-based shared-whiteboard application, and an

audio-connection facility, whereby users with LAN-based clients can hear other users whose AUAs are in the same virtual room. We plan to add similar support for video, and for large data objects such as medical images.

We continue to work on enhancements of our windowing model to give the AUA more control over the user's display. Currently, the remote-control applet can open windows and detect when the user closes windows; a small amount of JavaScript also allows the applet to close windows itself. Further enhancements will give application objects more control over the size and appearance of their "examine" windows, and add support for multiple frames within an object's HTML view.

As Artefact systems are deployed more widely, we will also investigate scaling issues. There are practical limits on the number of objects displayed in a room, the number of AUAs that can participate in a single linear thread of conversation, and the speed at which events can be propagated to a single browser. While our current applications bring users together in small groups, and thus remain well within those limits, techniques like those presented in [6] and [2] should allow us to support much larger numbers of simultaneously interacting users, for example in a "virtual auditorium".

REFERENCES

1. Activerse, Inc. *Ding! User's Guide*. Available from `http://www.activerse.com/ding/v10/docs/intro.htm`.
2. S. Benford, C. Greenhalgh, and D. Lloyd. Crowded collaborative virtual environments. In *Proceedings of ACM CHI '97 Conference*, pages 59–66., Atlanta, GA USA, 1997.
3. P. Dewan and R. Choudhary. Flexible user interface coupling in collaborative systems. In *Proceedings of ACM CHI '91 Conference*, pages 41–49, New Orleans, LA, USA, 1991.
4. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol—http/1.1. IETF Request for Comments No. 2068, January 1997.
5. N. T. C Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of ACM UIST '96*, pages 1–10, Seattle, WA, USA, 1996.
6. R. W. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rasmussen. Corona: A communication service for scalable, reliable group collaboration systems. In *Proceedings of ACM CSCW '96*, pages 140–140, Cambridge, MA, USA, 1996.
7. ichat, Inc. *Real-time Object Oriented Multimedia Server Administrator's Guide*, 2.1 edition, 1996.
8. Instinctive Technology, Inc. *eRoom*. Available from `http://www.instinctive.com`.
9. J. H. Lee, A. Prakash, T. Jaeger, and G. Wu. Supporting multi-user, multi-applet workspaces in CBE. In *Proceedings of ACM CSCW '96*, pages 344–353, Cambridge, MA, USA, 1996.
10. B. MacIntyre and S. Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proceedings of ACM UIST '96*, pages 83–94, Seattle, WA USA, 1996.
11. Mirabilis Ltd. *ICQ*. Available from `http://www.mirabilis.com`.
12. National Center for Supercomputing Applications. *The NCSA Habanero User's Guide*. Available from `http://www.ncsa.uiuc.edu/SDG/Software/Habanero /Docs/Environment/`.
13. Netopia, Inc. *Netopia Virtual Office*. Available from `http://www.netopia.com/software /nvo /win /overview.html`.
14. D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High latency, low bandwidth windowing in the jupiter collaboration system. In *Proceedings of UIST '95*, pages 111–120, Pittsburgh, PA, November 1995.
15. R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*, pages 297–312. John Wiley and Sons, 1996.
16. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
17. J. F. Patterson, R. D. Hill, S. L. Rohall, and W. S. Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *Proceedings of the Third Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 317–328, Los Angeles CA USA, 1990.
18. A. Prakash and H. S. Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of ACM CSCW '94*, pages 153–164, Chapel Hill, NC, USA, 1994.
19. M. Roseman and S. Greenberg. Building real time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.
20. M. Roseman and S. Greenberg. TeamRooms: network places for collaboration. In *Proceedings of CSCW '96*, pages 325–333, Cambridge, MA, 1996.
21. M. Sohlenkamp and G. G. Chwelos. Integrating communication, cooperation, and awareness: the DIVA virtual office environment. In *Proceedings of ACM CSCW '94*, pages 331–343, Chapel Hill, NC USA, 1994.

22. P. J. Spellman, J. N. Mosier, L. M. Deus, and J. A. Carlson. Collaborative virtual workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'97)*, pages 197–203, Phoenix, AZ, USA, 1997.

23. TeamWave Software Ltd. *TeamWave Workplace*. Available from `http://www.teamwave.com/`.