April 15, 1996

SRC Research Report 141a

Distributed Active Objects

Marc H. Brown and Marc A. Najork

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Distributed Active Objects

Marc H. Brown and Marc A. Najork

April 15, 1996

Publication History

This report will appear in the proceedings of the Fifth International World Wide Web Conference, May 6–10, 1996, `http://www5conf.inria.fr/`.

Abstract

Many Web browsers now offer some form of *active objects*, written in a variety of languages, and the number and types of active objects are growing daily in interesting and innovative ways. This report describes our work on Oblets, active objects that are distributed over multiple machines. Oblets are written in Obliq, an object-oriented scripting language for distributed computation. The high-level support provided by Oblets makes it easy to write collaborative and distributed applications.

1 Overview

One of the most exciting recent developments in Web-browser technology is *active objects*, where the browser downloads a program, executes it, and displays the program's user interface in a Web page. Sun's HotJava browser with Java applets pioneered active objects, showing Web pages with a wide range of content, from bouncing balls to spreadsheets to simulated science experiments. Many browsers now offer some form of active objects, written in a variety of languages.

This report describes *distributed active objects*, that is, active objects that can communicate with other active objects located on different machines across the Internet. High-level support for distributed computation makes it easy to write groupware, computer-supported cooperative work (CSCW) applications, and multi-player games as active objects.

Our environment for writing distributed active objects is based on Obliq [Cardelli95], an objected-oriented scripting language that was specifically designed for constructing distributed applications in a heterogeneous environment. We call active objects written in Obliq *Oblets* (**Ob**liq app**lets**). We have also built a family of Web browsers (DeckScape [Brown94], WebCard [Brown95], and WebScape) capable of running Oblets.

Obliq supports distributed computation by implementing all objects as *network objects* [Birrell93]. The methods of a network object can be invoked by other processes, in addition to the process that created the object. The initial connection between two processes occurs when one process registers an object with a name server under a unique name, and another process subsequently imports the object from that name server. Once the connection is established, network objects can be passed to other processes just as simply as passing any other type of data.

For network objects, method calls and field accesses have the same syntax regardless of where the object resides. It might reside in the same address space as the caller, or in a different address space either on the caller's machine or on some other (possibly different type of) machine. Thus, from a programmer's perspective, there is no difference between local and remote objects. As a result, network objects provide a uniform way for communication among Oblets, regardless of whether the Oblets are on the same Web page or on different Web pages displayed by different browsers on different machines. Moreover, network objects communicate directly, without server intervention. Thus, Oblets do not impose any load on an HTTP server, nor does a heavily loaded server affect their performance.

The rest of this report consists of four sections with increasingly complex examples, followed by a review of related work. The next section introduces fundamental concepts by showing a simple, non-distributed Oblet for adding two numbers. Section 3 shows the basics of distribution by developing a two-person game of tic-tac-toe. Section 4 shows a prototypical CSCW application—a chat room. The chat room allows an arbitrary number participants. The final example, Section 5, shows how to coordinate several different Oblets by developing a multi-view animation of an algorithm.

2 A Simple Oblet

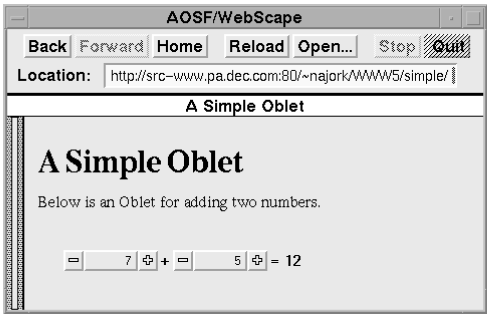
An Oblet is an Obliq program that defines a variable named `oblet`. This variable must contain an Obliq object with at least two fields: `vbt` and `run`. The `vbt` field is bound to a widget that will be installed on the screen when the Web page containing the Oblet is loaded. The `run` field is bound to a method that is invoked just after the `vbt` field is evaluated.

Oblets are placed into HTML documents via `insert`, an HTML tag proposed by the World Wide Web Consortium (W3C) for inserting multimedia objects into HTML3 pages [HTML3]. The markup for putting the Oblet at URL `foo.obl` into a document is:

```
<insert code="foo.obl" type="application/x-oblet"> </insert>
```

The `insert` tag also supports a variety of standard attributes, such as suggested dimensions, border size, and alignment. If suggested dimensions are not specified, the preferred dimensions of the widget contained in the Oblet's `vbt` field are used.

The following screen dump shows a simple Oblet for adding two numbers:



The user interface for that Oblet, defined by a FormsVBT s-expression [Avrahami89], is stored in the file `adder.fv`:

```
(HBox
  (Numeric %num1)
  (Text "+")
  (Numeric %num2)
  (Text "=")
  (Text %sum "0"))
```

A user interface in FormsVBT is a hierarchical arrangement of *components*. These include passive visual elements (e.g., `Text`), basic interactors (e.g., `Numeric`), modifiers that add interactive behavior to other components (e.g., `Button`), and layout operators that organize other components geometrically (e.g., `HBox`). Components can be further categorized as a split, filter, or leaf, based on the number of child components they support. A split can have any number of children (e.g., `HBox`), a filter has exactly one child (e.g., `Border`), and a leaf has no children (e.g., `Text`).

A component in FormsVBT can be given a name so that its attributes can be queried and modified at runtime. Names are also used for attaching callback procedures to interactors. In this Oblet, the two `Numeric` interactors are named `num1` and `num2`, and the `Text` component where the sum will be displayed is named `sum`.

The source for this Oblet is as follows:

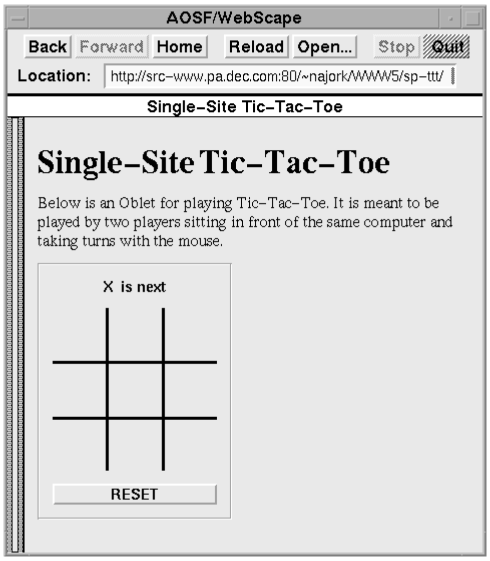
```
let doAdd =
  proc (fv)
    let n1 = form_getInt (fv, "num1");
    let n2 = form_getInt (fv, "num2");
    form_putText (fv, "sum", fmt_int (n1+n2))
  end;

let oblet = {
  vbt => form_fromURL (BaseURL & "adder.fv"),
  run =>
    meth (self)
      form_attach (self.vbt, "num1", doAdd);
      form_attach (self.vbt, "num2", doAdd);
    end
};
```

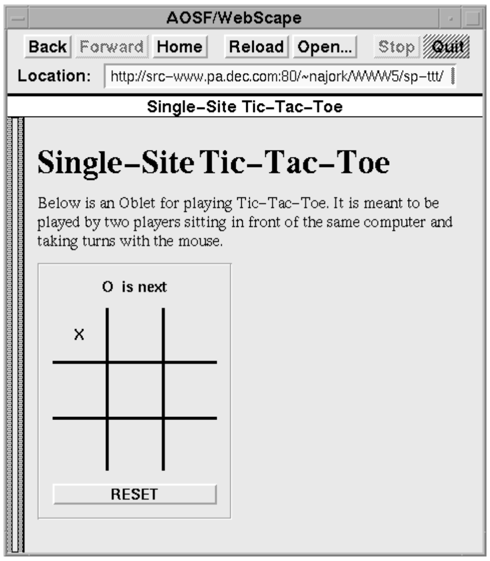
This Obliq program defines two variables: `doAdd` and `oblet`. Variable `doAdd` is a procedure that retrieves the values of both numeric interactors, and stores their sum in the component named `sum`.

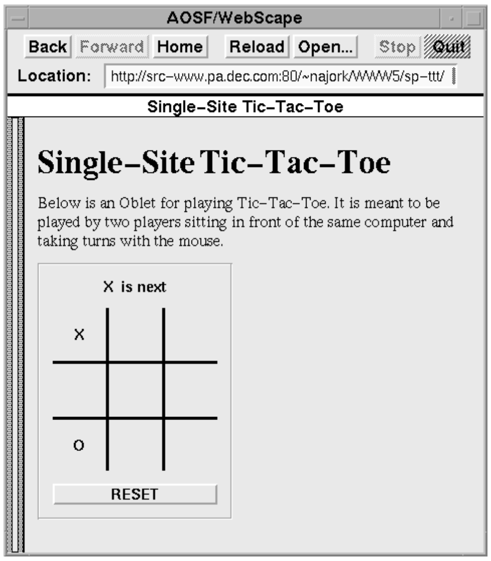
Variable `oblet` is an object with two fields, `vbt` and `run`. The `vbt` field is bound to a *form*, a widget that displays a FormsVBT s-expression. The procedure `form_fromURL` takes a URL as an argument and returns a form whose description is stored at this URL. The global variable `BaseURL` is the Oblet's absolute URL up through the last slash. The `run` method in this Oblet just attaches the callback procedure `doAdd` to the two numeric interactors. This procedure will be invoked whenever the user clicks on the plus or minus buttons of either interactor, or types a number into the editing field between the buttons. The form in which the event occurred is passed as an argument to the callback procedure. Recall that when the Web page containing this Oblet is loaded, the `vbt` field will be evaluated and the result displayed on the page, the `run` method will be invoked, and finally the page will become visible.

3 A Distributed Game Oblet

This section describes an Oblet for playing tic-tac-toe. We'll first develop a single-site game; then, we'll show how to extend this game to two sites. The following screen dumps show the first three moves in the single-site game:







The FormsVBT description for this Oblet contains a message line that indicates whose turn it is, a game grid consisting of nine squares, and a "RESET" button at the bottom that is used to clear the squares. The message line is a `Text` component named `status`. Each square of the game grid consists of a `Button` and a `Text` component. The `Button` components are named `btn1`, ..., `btn9`, and the `Text` components are named `lab1`, ..., `lab9`. The "RESET" button is named `reset`. Finally, the form's top-level component has the name `board`.

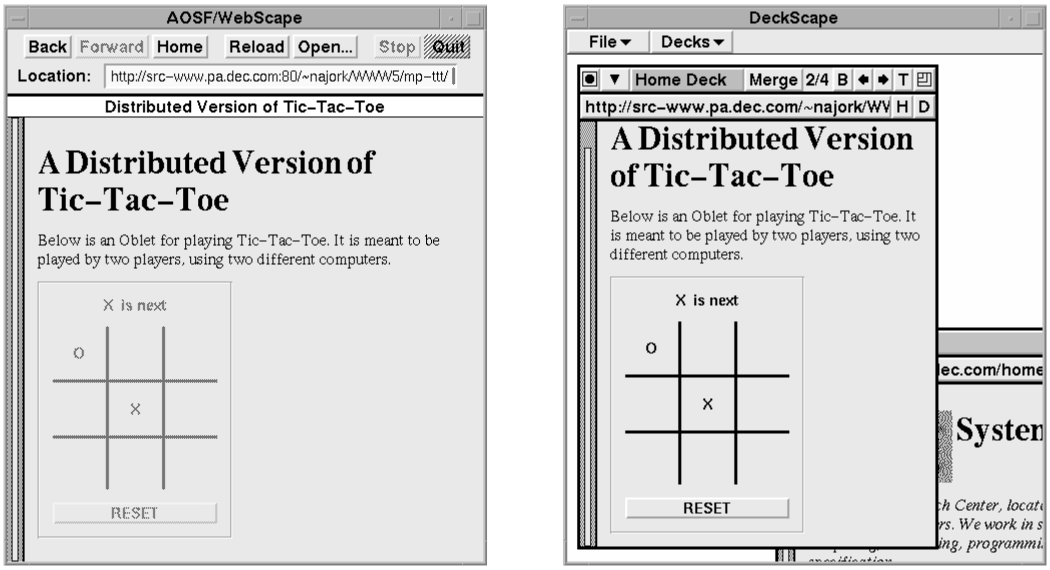
The code for the Oblet is as follows:

```
let otherPlayer =
  proc (p)
    if p is "X" then "O" else "X" end
  end;

let oblet = {
  vbt => form_fromURL (BaseURL & "tic-tac-toe.fv"),
  c   => ok,

  reset =>
    meth (self)
      for i = 1 to 9 do
        form_putText (self.vbt, "lab" & fmt_int(i), "");
      end;
    end,

  move =>
    meth (self, label, player)
      form_putText (self.vbt, label, player);
      form_putText (self.vbt, "status", otherPlayer(player) & " is next");
    end,

  nextTurn =>
    meth (self)
      self.c := otherPlayer(self.c);
    end,

  run =>
    meth (self)
      self.c := "X";

      let doReset =
        proc(fv)
          self.reset ();
        end;

      let doPress =
        proc (m)
          let label = "lab" & fmt_int(m);
          if form_getText (self.vbt, label) is "" then
            self.move (label, self.c);
            self.nextTurn ();
          end;
        end;

      form_attach (self.vbt, "reset", doReset);
      for i = 1 to 9 do
        let p = proc(fv) doPress(i) end;
        form_attach (self.vbt, "btn" & fmt_int(i), p)
      end;
    end
};
```

This Oblet, in addition to the required `vbt` field and `run` method, also has a field `c` and methods `reset`, `move`, and `nextTurn`. The field `c` will be a string indicating the player about to move, either “X” or “O”. The `reset` method clears the label displayed in each square of the game grid. The `move` method stores the string `player` into the `Text` component whose name is `label`, and also updates the message line to indicate whose turn is next. The `nextTurn` method changes whose turn it is, that is, it changes the value of the field `c`. The last two methods use the procedure `otherPlayer`, which takes one player’s symbol and returns his opponent’s symbol.

The body of the `run` method initializes field `c`, and then attaches callback procedures to the various interactors on the board. Procedure `doReset` is attached to the “RESET” button; it will invoke the `reset` method of the object `oblet`. A procedure `p` is attached to each of the nine buttons, `btn1`, ..., `btn9`. This procedure effectively captures the value of `i`, the index of each square on the game grid. When `p` is invoked (in response to a user clicking in a square), it calls procedure `doPress(i)`, which checks that the square is empty, and if so, invokes the Oblet’s `move` and `nextTurn` methods.

We now convert the single-site version of tic-tac-toe into a two-site, distributed version. The following figure shows a snapshot of a two-site game in progress. The left image shows the browser (WebScape) used by player “O”, the right image shows the browser (DeckScape) used by player “X”. The message line indicates that player “X” is next, and the Oblet of player “O” is grayed out, indicating that it is non-responsive for the time being.



The changes to the Oblet code are remarkably simple. First, we extend the `oblet` to include an extra field, `opp`, which is the `oblet` of the opponent. Second, we use the field `c` in a slightly different way: In the single-site version, `c` was a string that indicated whose turn it was; it changed after each turn. In the two-site version, it is also a string, but it never changes. Rather, it is initialized to the player in whose browser the Oblet is run. Finally, there are changes to the `nextTurn` and `run` methods. Here is the entire Oblet, with unchanged parts elided:

```
let otherPlayer = ...;

let oblet = {
  vbt      => ...,
  c        => ok,
  opp      => ok,
  reset    => ...,
  move     => ...,

  nextTurn =>
    meth (self)
      if form_getReactivity(self.vbt, "board") is "active" then
        form_putReactivity(self.vbt, "board", "dormant");
      else
        form_putReactivity(self.vbt, "board", "active");
      end;
    end,

  run =>
    meth (self)
      try
        self.opp := net_import ("TicTacToe", "ash.pa.dec.com");
        self.opp.opp := self;
        self.c := "X";
      except net_failure =>
        net_export ("TicTacToe", "ash.pa.dec.com", self);
        form_putReactivity (self.vbt, "board", "dormant");
        self.c := "O";
      end;

      let doReset =
        proc(fv)
          self.reset ();
          self.opp.reset ();
        end;

      let doPress =
        proc (m)
          let label = "lab" & fmt_int(m);
          if form_getText (self.vbt, label) is "" then
            self.move (label, self.c);
            self.opp.move (label, self.c);
            self.nextTurn ();
            self.opp.nextTurn ();
          end;
        end;

      form_attach (self.vbt, "reset", doReset);
      for i = 1 to 9 do
        let p = proc(fv) doPress(i) end;
        form_attach (self.vbt, "btn" & fmt_int(i), p)
      end;
    end
};
```

We start a game by visiting the tic-tac-toe Web page, which causes the tic-tac-toe Oblet to be loaded and its `run` method to be invoked. The first part of the `run` method attempts to import an object called `TicTacToe` from the name server at machine `ash.pa.dec.com`. This call succeeds if there already is a player waiting for a game to begin. In this case, the opponent's `oblet` is stored in our `opp` field, our `oblet` is stored in our opponent's `opp` field, and we choose "X" to be our symbol. If the attempt to import `TicTacToe` fails, then we export our `oblet` to the name server at `ash.pa.dec.com`, make our game board dormant (i.e., grayed out and unresponsive to mouse activity), and choose "O" as our symbol. For the sake of simplicity, we ignore the race condition of more than one player executing this code simultaneously.

The change to the `doReset` callback is simple: we invoke the `reset` method not only on our `oblet`, but also on our opponent's `oblet`. The change to the `doPress` callback is similar: rather than invoking `move` and `nextTurn` only on our `oblet`, we also invoke these methods on our opponent's `oblet`. The rest of the `run` method is unchanged: callbacks are attached to the interactors.

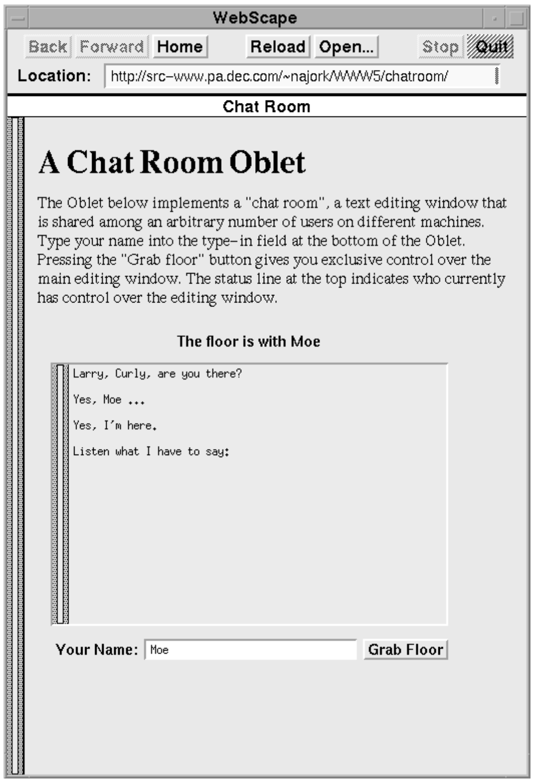
The final change in the Oblet is to the `nextTurn` method. In the single-site version, we changed the value of field `c` from "X" to "O" and vice versa. Here, we change the reactivity of the game board from active to dormant and vice versa. Therefore, each player can press a button only when it is his turn to move.

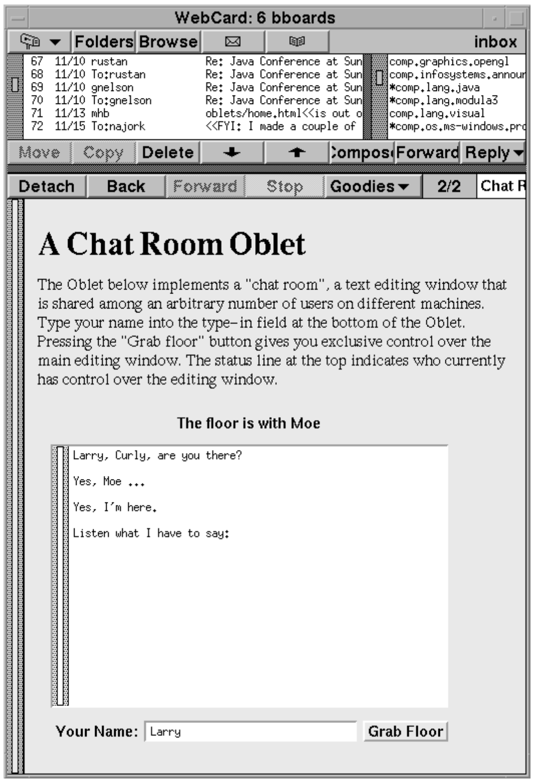
It is worth emphasizing that `self.opp` denotes an object that resides on the opponent's machine. This implies that the assignment to `self.opp.opp` and the execution of the `self.opp.reset`, `self.opp.move`, and `self.opp.nextTurn` method calls take place on this other machine.

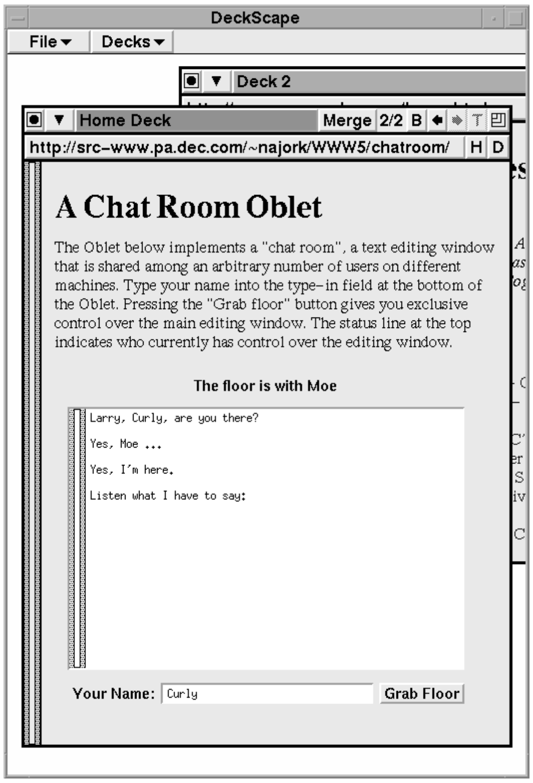
4 A Distributed Chat Room Oblet

Oblets are flexible enough to allow distributed computations to have arbitrary topologies. In the tic-tac-toe example, we had two `oblet` objects performing peer-to-peer communication. In this example, we use a star topology to implement a multi-person chat room. At the center of the star, we have a *conference control* object; at the periphery are the Oblets belonging to the participants. When a user types into his chat room Oblet, it informs the conference controller of the new text, which then relays the update to all the participating Oblets; in other words, Oblets do not communicate with other Oblets directly. Our chat room also provides a mechanism for floor control.

The following three images show the chat room Oblet running in different browsers (WebScape, WebCard, and DeckScape). Each browser is running on a different machine. The participants in the chat room are Moe, Larry, and Curly. Currently the floor is with Moe, as indicated by the status line over the editing region and by the color of the editing region in Moe's browser.


WebScape
Back Forward Home Reload Open... Stop Quit
Location: http://src-www.pa.dec.com/~najork/WWW5/chatroom/
Chat Room
A Chat Room Oblet
The Oblet below implements a "chat room", a text editing window that is shared among an arbitrary number of users on different machines. Type your name into the type-in field at the bottom of the Oblet. Pressing the "Grab floor" button gives you exclusive control over the main editing window. The status line at the top indicates who currently has control over the editing window.
The floor is with Moe
Larry, Curly, are you there?
Yes, Moe ...
Yes, I'm here.
Listen what I have to say:
Your Name: Moe
Grab Floor


WebCard: 6 bboards
Folders Browse
inbox
Move Copy Delete Forward Reply
Detach Back Forward Stop Goodies 2/2
A Chat Room Oblet
The Oblet below implements a "chat room", a text editing window that is shared among an arbitrary number of users on different machines. Type your name into the type-in field at the bottom of the Oblet. Pressing the "Grab floor" button gives you exclusive control over the main editing window. The status line at the top indicates who currently has control over the editing window.
The floor is with Moe
Larry, Curly, are you there?
Yes, Moe ...
Yes, I'm here.
Listen what I have to say:
Your Name: Larry
Grab Floor


DeckScape
File Decks
Deck 2
Home Deck
Merge 2/2 B T
http://src-www.pa.dec.com/~najork/WWW5/chatroom/
H D
A Chat Room Oblet
The Oblet below implements a "chat room", a text editing window that is shared among an arbitrary number of users on different machines. Type your name into the type-in field at the bottom of the Oblet. Pressing the "Grab floor" button gives you exclusive control over the main editing window. The status line at the top indicates who currently has control over the editing window.
The floor is with Moe
Larry, Curly, are you there?
Yes, Moe ...
Yes, I'm here.
Listen what I have to say:
Your Name: Curly
Grab Floor

Here is the FormsVBT s-expression for the chat room Oblet:

```
(Rim (Pen 10)
  (VBox
    (Text %floorWith "The floor is free right now")
    (Glue 10)
    (Shape (Width 300) (Height 200)
      (Frame Lowered
         (Filter Passive
           (TextEdit (BgColor "White") %mainEditor))))
    (Glue 10)
    (HBox
      (Text "Your Name:")
      (Frame Lowered (TypeIn (BgColor "White") %myName))
      Fill
      (Button %grabFloor "Grab Floor"))))
```

The `floorWith` component is the message line above the large editing region; it will contain a message indicating who owns the floor. The `mainEditor` is the large (300x200) editing region. The `Filter` component surrounding the region is used to set the reactivity of the region; in the passive state, the region is unresponsive to mouse and keyboard activity, but it is not grayed out, as it would be in the dormant state. The type-in field where each participant identifies himself is named `myName`. Finally, the "Grab Floor" button has been given the name `grabFloor`.

As we shall see, callback procedures will be attached to the "Grab Floor" button and to the large editing region. When the user clicks on the "Grab Floor" button, the message line on all participating Oblets will indicate who owns the floor (using the contents of the type-in field of the Oblet now owning the floor), the editing region on all Oblets (other than the one owning the floor) will become passive, and the editing region in the Oblet owning the floor will become active and its color will change to pink. When the user who owns the floor types a keystroke into the editing region, all of the participating Oblets will be notified of the updated text.

Recall that Oblets do not communicate with other Oblets directly. Rather, they use a conference control object to report the changes, and this object then relays the changes to the other Oblets. Here is the definition of the conference control object:

```
let ProtoConfControl = {
  oblets => [],
  onFloor => ok,
  contents => "",

  register =>
    meth (self, oblet)
      self.oblets := self.oblets @ [oblet];
      oblet.updateText (self.contents);
      if self.onFloor isnot ok then
        oblet.transferFloor (self.onFloor);
      end;
    end,

  transferFloor =>
    meth (self, name)
      self.onFloor := name;
      foreach o in self.oblets do
        o.transferFloor (name);
      end;
    end,

  updateText =>
    meth (self, contents)
      self.contents := contents;
      foreach o in self.oblets do
        o.updateText (contents);
      end;
    end
};
```

The `oblets` data field is an array of the Oblets that have registered themselves with the conference control object. Each element of this array is an `oblet` that typically resides on a different machine. The `onFloor` data field is the name of the user who currently has the floor, and the `contents` data field contains the current contents of the editing region. These two fields are needed in order to initialize the display of a new participant entering this chat room.

The `register` method will be called by a new Oblet `oblet` when it is initialized, as part of its `run` method. The new Oblet is appended to the `oblets` array, and then it is notified both of the current contents of the editing region and of the owner of the floor, if there is one.

The `transferFloor` method will be called by an Oblet when the user clicks on the "Grab Floor" button. This method stores in `onFloor` the name of the user who now owns the floor, and then iterates through all of the Oblets in the conference, invoking the `transferFloor` method on each Oblet to inform it of the new floor owner.

Finally, the `updateText` method will be called on each keystroke by the Oblet that owns the floor, passing in the current contents of the editing region. (Passing just the keystroke is not sufficient, since a single character could result in various editing actions, depending on the key bindings used by the Oblet.) The `updateText` method stores in `contents` the new contents of the editing region and then updates all of the Oblets in the chat room by invoking the `updateText` method on each one.

We are now ready to examine the code for the Oblet:

```
let oblet = {
  vbt => form_fromURL (BaseURL & "chatroom.fv"),

  transferFloor =>
    meth (self, name)
      form_putReactivity (self.vbt, "mainEditor", "passive");
      form_putBgColor (self.vbt, "mainEditor", color_named("white"));
      form_putText (self.vbt, "floorWith", "The floor is with " & name);
    end,

  updateText =>
    meth (self, contents)
      form_putText (self.vbt, "mainEditor", contents);
    end,

  run =>
    meth (self)
      var confControl = ok;
      try
        confControl := net_import("ConfControl", "ash.pa.dec.com");
      except net_failure =>
        confControl := ProtoConfControl;
        net_export("ConfControl", "ash.pa.dec.com", confControl);
      end;

      let doGrabFloor =
        proc (fv)
          confControl.transferFloor (form_getText (fv, "myName"));
          form_putReactivity (fv, "mainEditor", "active");
          form_putBgColor (fv, "mainEditor", color_named("pink"));
        end;

      let doKeyEvent =
        proc (fv)
          confControl.updateText (form_getText (fv, "mainEditor"));
        end;

      confControl.register (self);
      form_attach (self.vbt, "grabFloor", doGrabFloor);
      form_attach (self.vbt, "mainEditor", doKeyEvent);
    end
};
```

The Oblet defines two methods, `transferFloor` and `updateText`; as we just saw, these methods will be invoked by the conference control object in response to a user in an arbitrary Oblet in the chat room grabbing the floor or typing into the editing region, respectively. These methods are straightforward: the `transferFloor` method makes the editing region passive and sets its background to be white, and then updates the message line. The `updateText` message changes the contents of the editing region.

The Oblet's `run` method first contacts the name server on the machine `ash.pa.dec.com` to obtain a conference control object registered under the name `ConfControl`. If there is such an object, it is stored in the variable `confControl`. Otherwise, a new conference control object is registered with the name server and also stored in `confControl`. As in the tic-tac-toe example, we do not show the code necessary for preventing the race condition of several users executing the `try-except` statement simultaneously. After defining callback procedures `doGrabFloor` and `doKeyEvent`, this Oblet registers itself with the conference controller, and finally attaches the callback procedures to the "Grab Floor" button and to the editing region.

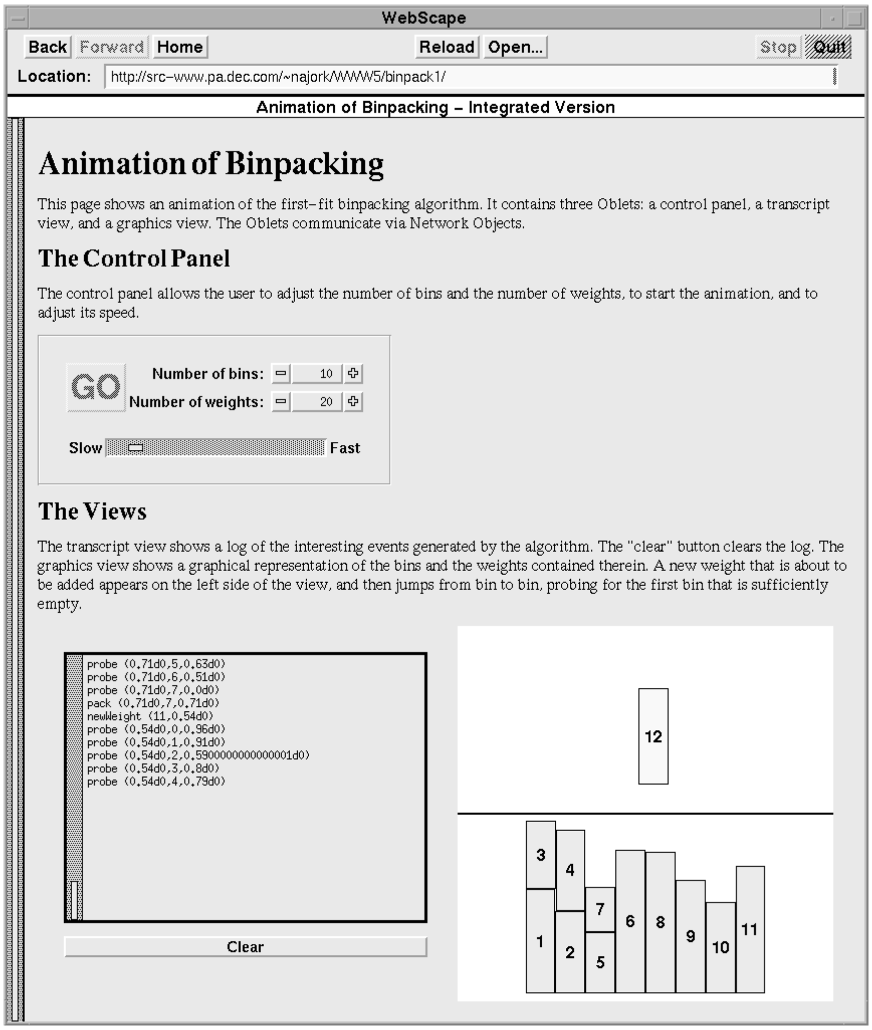
The `doGrabFloor` callback procedure invokes the `transferFloor` method on the `confControl` object (which then calls the `transferFloor` method on all Oblets in the chat room, including this one), and then makes its own editing region active and colored pink. The `doKeyEvent` callback procedure simply invokes the `updateText` method on the `confControl` object, passing to it the text in the editing region.

Again, it is important to point out that invoking a method `m` on the `confControl` object is done just by calling `confControl.m()`, regardless of where the `confControl` object resides. In this example, the conference control object will be local to the Oblet that creates it, and remote to all other Oblets.

There are many features that could be added to the chat room in a fairly straightforward way. For example, it would be nice to be able to prevent another user from taking away the floor, to allow users to leave the chat room, to create new chat rooms, to see existing chat rooms, to handle exceptions that might result from network partitions, and so on. In addition, one can easily imagine more efficient implementations, such as reporting only changes to the editing region rather than reporting the region's entire contents after each keystroke.

5 Oblets for Algorithm Animation

Obliq's network objects provide a uniform and elegant way for objects to communicate, regardless of the address space they exist in and the machine they reside on. The two previous examples showed the obvious use for network objects: to communicate among objects on different machines. The example in this section uses network objects to allow Oblets running in the same browser (on the same Web page or on different Web pages) to communicate. This could be achieved through simpler mechanisms; after all, all Oblets on the same browser are in the same address space. However, network objects minimize the number of concepts needed by a programmer, since they handle this case in the exact same way as the distributed case. Moreover, network objects make it easy to reuse Oblets in distributed settings without any code changes.



This example uses network objects to coordinate multiple Oblets in the domain of algorithm animation [Brown84]. A typical algorithm animation system has a control panel and a collection of views, each in its own window. The control panel is used for specifying data, starting the algorithm, controlling the animation speed, and so on. In order to animate an algorithm, strategically important points of its code are annotated with procedure calls that generate *interesting events*. These events are reported to the algorithm animation system, which in turn forwards them to all interested views. Each view responds to interesting events by updating its display appropriately.

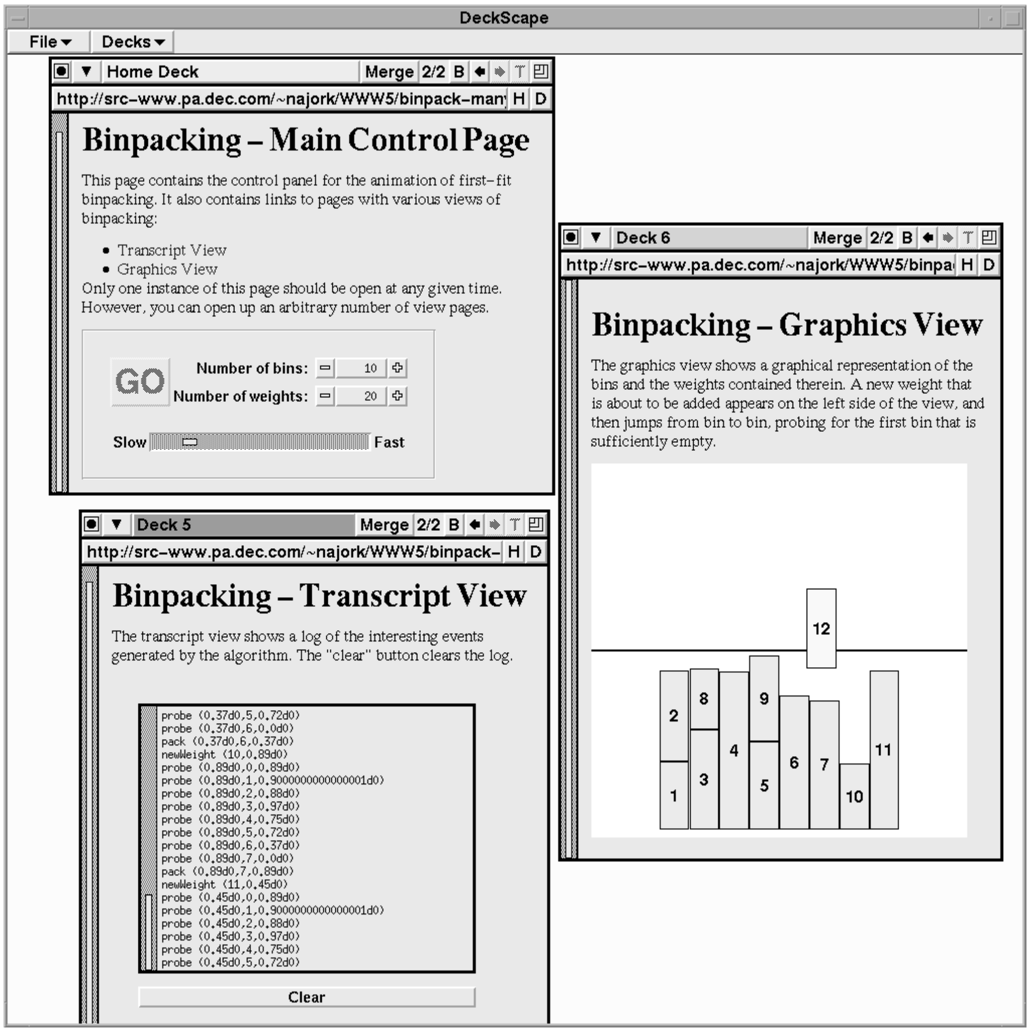
The screen dump on the previous page shows an animation of first-fit binpacking. The control panel and the views are implemented by separate Oblets.

We use an *event manager* object, similar to the conference control object in the chat room example, to relay interesting events from the algorithm to the views. For each interesting event there is a corresponding method both in the event manager object and in each view Oblet. When an interesting event occurs, the algorithm Oblet invokes the corresponding method of the event manager object, which in turn relays the event to each view. Typically, views react by showing some animation reflecting the changes in the program. In order for the animation in the views to happen simultaneously, the event manager forks a thread for each registered view, the thread calls the view's method corresponding to the interesting event, and the event manager waits until all of the threads have completed before returning to the algorithm.

For example, when a binpacking algorithm is trying to insert a particular weight `w` into a bin `b` that already contains a number of weights totaling up to `amt`, it calls `z.probe(w,b,amt)`. The `probe` method of the event manager object `z` is implemented as follows:

```
let z = {
  views => [],
  ...
  probe =>
    meth (self,w,b,amt)
      let threads =
        foreach v in self.views map
          let closure = proc() v.probe(w,b,amt) end;
          thread_fork(closure)
        end;
      foreach t in threads do
        thread_join(t)
      end;
    end;
    ...
};
```

The screen dump on the previous page showed the Oblets for the control panel and each view all on the same Web page. However, there is no need for the Oblets to be located on the same page. In fact, if we put each Oblet on a separate page, the user can dynamically select the set of views visible (or even have more than one copy of any view). In the screen dump on the following page, the Web page containing the control panel has links for pages containing the various views. Clicking on such a link brings up a page for the view, which the DeckScape browser can optionally display in a separate window.



At first blush, it would appear that this example uses network objects merely for the coding elegance they offer, rather than for any of their distributed aspects. That is, in the two screen dumps in this section, all of the Oblets exist in the same address space, namely that of the browser. However, because Oblets are network objects, we have far more flexibility. For instance, we can use the Oblets — *without any changes* — in an Electronic Classroom setting. In such a setting, the instructor and all students run Deckscape on their individual machines (using the same name server). The instructor uses the control page Oblet to drive the animation, and each student sees a set of views portraying the workings of the algorithm. This scenario is explored in depth elsewhere [Brown96].

6 Related Work

Oblets bring together active objects and distributed computation. The best known language for active objects is Java [Java]. HotJava was the first browser to support Java applets; in the meantime support for Java applets has been integrated into Netscape Navigator. Most major commercial browser vendors have subsequently announced intended support for Java applets.

The most serious potential competitor to Java-based browsers is probably Microsoft's Internet Explorer, which plans to integrate support for active objects written in Visual Basic (as well as for those written in Java) [Microsoft]. However, the current version of Internet Explorer does not support active objects.

In the research community, a number of browsers have been developed that support other languages for writing active objects. Most of these browsers are written in interpreted languages and support active objects written in the same language. Examples include Hush [vanDoorn95] and SurfIt! [SurfIt!], implemented in Tcl/Tk; MMM [MMM], implemented in CAML/Tk; and Grail [Grail], implemented in Python.

None of the browsers and languages mentioned above has any high-level support for distributed programming. However, the HORB system [HORB] adds the equivalent of network objects to Java. It consists of a name server and a compiler that creates network object classes based on Java interface specifications. Unlike Obliq, HORB is a first-order language, meaning that only data, but not computations, can be migrated over the network. Also, HORB does not provide distributed garbage collection.

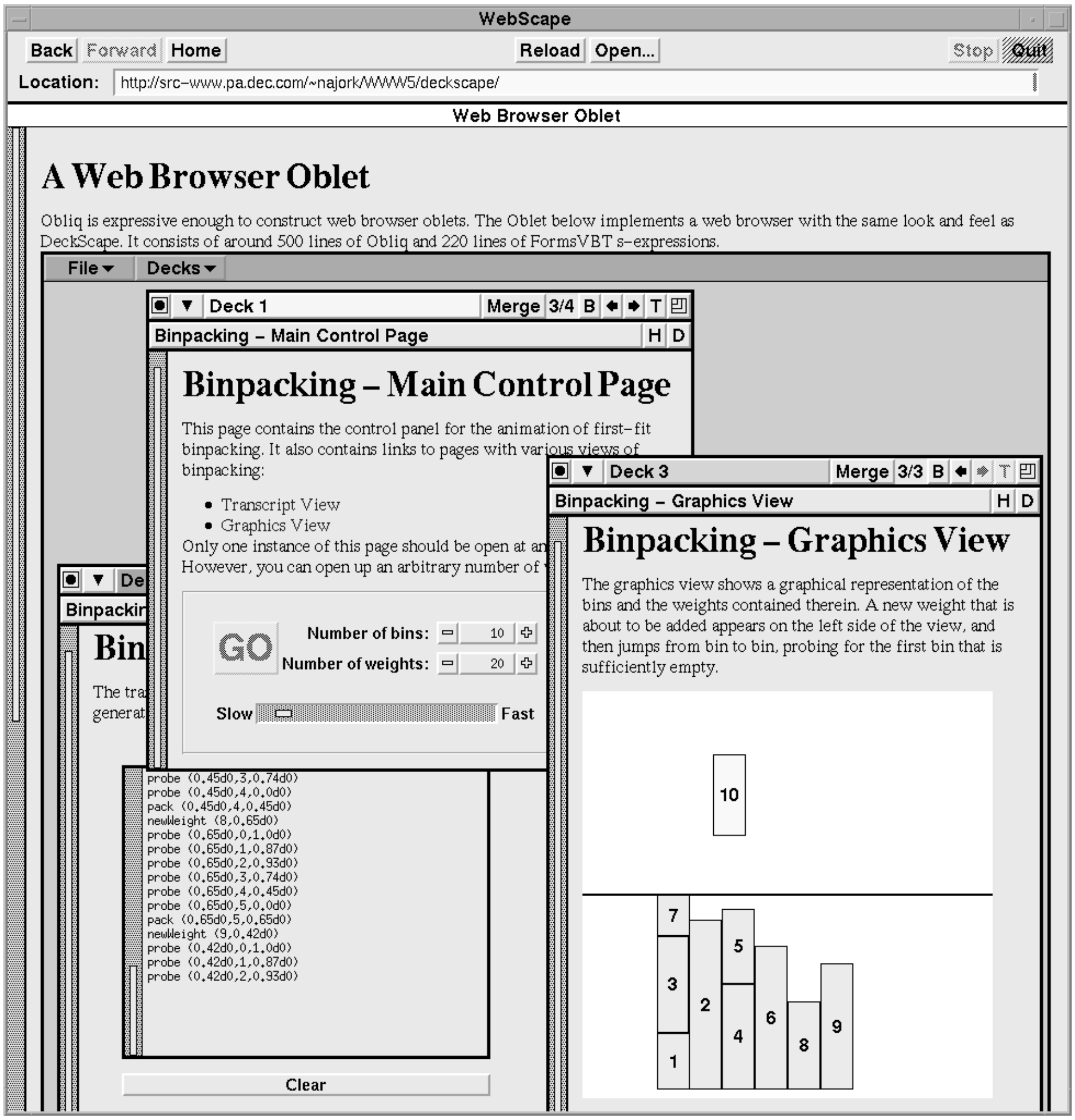
Obliq [Cardelli95] is a lexically-scoped language that supports distributed object-oriented computation. It has been integrated into commercial Web browsers by defining an Obliq MIME type and configuring the browser to use the Obliq interpreter as an external viewer [Bharat95]. Many other distributed languages exist, commercially (e.g., General Magic's Telescript [Telescript]) and in academia (e.g., Orca [Bal92]). However, we are not aware of any such language having been integrated with a Web browser.

7 Conclusion

The example Oblets shown in this report have been small, for didactic reasons. However, Obliq is a full-strength programming language with access to a rich set of libraries, including multimedia objects and even Web pages.

The DeckScape browser below shows a "Virtual TV" Oblet; the main screen and each of the buttons show live video streams. New video streams can be added by typing the IP address of a video server into the type-in field.

The WebScape browser on the next page shows an Oblet that implements the look-and-feel of DeckScape, but uses a different color for the main canvas. Within this Oblet, we are visiting Web pages containing the various binpacking animation Oblets we saw before. This Oblet consists of about 500 lines of Obliq code and 200 lines of FormsVBT user-interface specification.



We have not explored the issues of security and fault tolerance, both very important and very real problems. In the area of security, Web browsers should be able to authenticate the origin of an Oblet and to protect the user against malicious Oblets. In the area of fault tolerance, Oblets should be able to gracefully handle disruption of network services and nonavailability of network resources.

Many analysts feel that two of the most important technology themes for the remainder of the decade are the Web and using computers for collaboration. Oblets provide an elegant programming framework for bringing collaborative and distributed applications to the Web.

References

[Avrahami89] Gideon Avrahami, Kenneth P. Brooks, Marc H. Brown.
A Two-View Approach To Constructing User Interfaces.
Computer Graphics, **23**(3):137–146, July 1989.

[Bal92] H.E.Bal, M.F.Kaashoek, and A.S.Tanenbaum.
Orca: A Language for Parallel Programming of Distributed Systems.
IEEE Transactions on Software Engineering, **18**(3):190–205, March 1992.

[Bharat95] Krishna Bharat and Luca Cardelli.
Distributed Applications in a Hypermedia Setting.
Proc. of the 1st Intl. Workshop on Hypermedia Design,
pages 185–192, June 1995.

[Birrell93] Andrew D. Birrell, Greg Nelson, Susan Owicki, and Edward P. Wobber.
Network Objects.
Proc. of the 14th ACM Symposium on Operating System Principles,
pages 217–230, December 1993.

[Brown84] Marc H. Brown and Robert Sedgewick.
A System for Algorithm Animation.
Computer Graphics, **18**(3):177–186, July 1984.

[Brown94] Marc H. Brown and Robert A. Shillner.
DeckScape: An Experimental Web Browser.
Computer Networks and ISDN Systems, **27**(1995) 1097–1104.

[Brown95] Marc H. Brown.
Browsing the Web with a Mail/News Reader.
Proc. of the 8th ACM Symposium on User Interface Software and Technology,
pages 197–198, November 1995.

[Brown96] Marc H. Brown and Marc A. Najork.
Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom.
Research Report #142, Digital Equipment Corporation Systems Research Center, Palo Alto, CA (May 1996).

[Cardelli95] Luca Cardelli.
A Language with Distributed Scope.
Computing Systems, **8**(1):27–59, January 1995.

[Grail] Grail Home Page.
`http://monty.cnri.reston.va.us/grail-0.2/`

[HTML3] HTML3 Linking and Embedding Model.
`http://www.w3.org/hypertext/WWW/TR/WD-insert-951221.html`

[HORB] HORB Home Page.
`http://ring.etl.go.jp/openlab/horb/`

[Java] Java: Programming for the Internet.
`http://java.sun.com/`

[Microsoft] Internet Development Toolbox.
`http://www.microsoft.com/INTDEV/`

[MMM] MMM Browser Home page.
`http://pauillac.inria.fr/~rouaix/mmm/`

[SurfIt!] SurfIt!
`http://pastime.anu.edu.au/SurfIt/`

[Telescript] Telescript.
`http://www.genmagic.com/Telescript/index.html`

[vanDoorn95] Matthijs van Doorn and Anton Eliëns.
Integrating Applications and the World-Wide Web.
Computer Networks and ISDN Systems, **27**(1995) 1105–1110.