ReadUp: A Widget For Reading

William C. Janssen

Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California, 94304 USA
janssen@parc.com

Abstract. User interfaces for digital library systems must support a wide range of user activities. They include search, browsing, and curation, but perhaps the most important is actual reading of the items in the library. Support for reading, however, is usually relegated to applications which are only loosely integrated with the digital library system. One reason for this is the absence of toolkit widget support for the activity of reading. Most user interface toolkits instead provide support for either text editing or text presentation. This makes it difficult to write applications which support reading well. In this paper we describe the origins, design, and implementation of a new Java Swing toolkit widget called *ReadUp*, which provides support for reading page images in a digital library application, and discuss briefly how it is being used.

1 Introduction

The UpLib personal digital library system [10] is designed for storage, organization, retrieval, and use of an individual's digital documents. As part of this system, we originally included a simple Web-based document reader which presented each page of the document being read as an image on a Web page, along with a sidebar that provided access to operations on the document. Page images were clipped to increase effective resolution, using the techniques described in [9]. The image had a superimposed HTML imagemap, so that clicking with the mouse on the right side of the page would turn to the next page by following a link to a Web page that had the image of the next page on it. Similarly, clicking on the left side of the page would take the reader to the previous page. On the side of the page image, small icons provided the page index number of the page, explicit following-page/preceding-page buttons, and an UpLib logo button which when clicked would take the user to the UpLib repository overview.

This document reader implementation had many deficiencies. On a fast machine, the page turn could happen quickly enough that users hadn't realized it *had* happened, and would frequently double or triple click, turning multiple pages inadvertently. The trail of pages cluttered the Web browser's history stack, making effective use of the "back" button difficult. There was no provision for annotation or markup of any kind, even simple bookmarks. There was no ability to search the text of the page or the document. Web browsers often ignored the

caching instructions provided in the HTTP headers, and cached potentially sensitive page images to disk. The sidebar was generated statically, used frames, and contained seldom-used page thumbnails, display of which was not synchronized with the particular large page image currently visible. Other clients for UpLib were being developed, with similar reading interfaces, and they shared some of these problems.

To address these deficiencies, we have implemented a reading widget as a Java Swing `JComponent`, which we use in both an applet form, and directly in other applications. We call it the *ReadUp* widget. In the rest of this paper, we describe the widget's design and interface, then discuss some implementation issues, and finally consider the ways in which it is being used.

2 Related Work

A number of studies of subjects reading both paper and virtual paper documents indicate several important needs for reading interfaces. In [13], looking at readers of both paper and virtual documents, the authors remark that "critical differences have to do with the major advantages that paper offers in supporting annotation while reading, quick navigation, and flexibility of spatial layout.", while in [14], they examine annotation more fully, citing the differences between making marks directly on the source and making notes about the source, finding both to have value. Marshall examines annotations in books in great detail in [12], stressing the importance of support for fluid, *in situ*, informal markings, integrated with the reading activity.

Navigation in digital documents is also an issue. The authors of [1] point out that linear reading is "an unrealistic characterisation of how people read in the course of their daily work", a finding reinforcing the ideas of reading in [2]. Another survey [7] suggests difficulty with page manipulation and navigation when reading from screen devices, compared to paper.

A common approach to this need for better support for annotation and navigation in digital documents is to write a separate reading application which supports page manipulation, search, and annnoation. Some of these applications, such as Lectrice [5] and XLibris [16], have been applications directly supporting annotation and navigation, but somewhat embedded in tablet PC hardware. Others, such as Lectk [5], Multivalent [15], 3Book [4], and "Open the Book" [6], also address the support issues, and run as applications on more conventional computing platforms. Building such an application gives full control over page presentation, and allows for additional experimentation. However, this approach does not support a Web-based interface, and has the further disadvantage of having to be installed on potential clients before any documents can be read with it. In addition, other client applications cannot take advantage of this functionality.

3 The ReadUp widget

We eventually realized that if the custom application was designed as a new toolkit widget, many of these problems could be addressed. It could be bundled up in either a Web browser plug-in or in an applet, and used to support the Web interface to UpLib. It could be extended to support experimental reading operations. Other applications could use the widget to support reading from an UpLib repository.

It is possible to write a browser plug-in *sui generis*, but it would have to be downloaded and installed before it could be used. Two existing plug-in technologies, Flash and Java, provide useful interface toolkits and are widely preinstalled. The use of Java seemed particularly appealing, despite the somewhat clumsy nature of the language, as much of the work in our group is being done in Java. The default user interface toolkit in Java, called Swing [20], is also a solid and portable, if unimaginative, base on which to work. Additionally, the Java applet is a well-known way to present client interfaces to users in Web pages.

A widget as complex as ReadUp contains a complete embedded user interface, which must be neutral enough to mesh well with the various applications it is used in. This interface, which perhaps should be thought of as the *reader* interface, is designed to be flexible enough to experiment with new modes of interaction for reading, but also in its basic form provide support for two major reading activities identified by the previously cited user studies of reading: annotation and document navigation. In addition, it supports two experimental reading modes, rapid serial visual presentation and animated dynamic highlighting.

Figure 1 shows a document open in a ReadUp widget; the widget itself may be embedded in a larger user interface, or simply in a stand-alone top-level window. The current page is displayed as an image in the center; either one or two pages (figure 2) may be displayed. On the side, a toolbar provides direct access to commonly used functions: page-turning, search, annotation, and bookmarks. Thin *page-edge* indicators at the top and bottom provide visual indication of where the reader is in the document. The toolbar and page-edge controls are optional; their display may be suppressed by the application. Both are "skinnable" so that their appearance can be adjusted by applications.

3.1 Document navigation

The variety of ways in which people manipulate paper pages and navigate through documents is noted continuously in studies of readers. Many of the previously mentioned reading systems have attempted to support this kind of manipulation and navigation for virtual documents. ReadUp brings support for these reading activities into the toolkit. First, it provides several ways to simply turn pages, and provides a page-turn animation to provide feedback to the user on the event. Second, it supports operations on the document as a whole through overviews, bookmarks, and page-edges. Third, it supports search through the

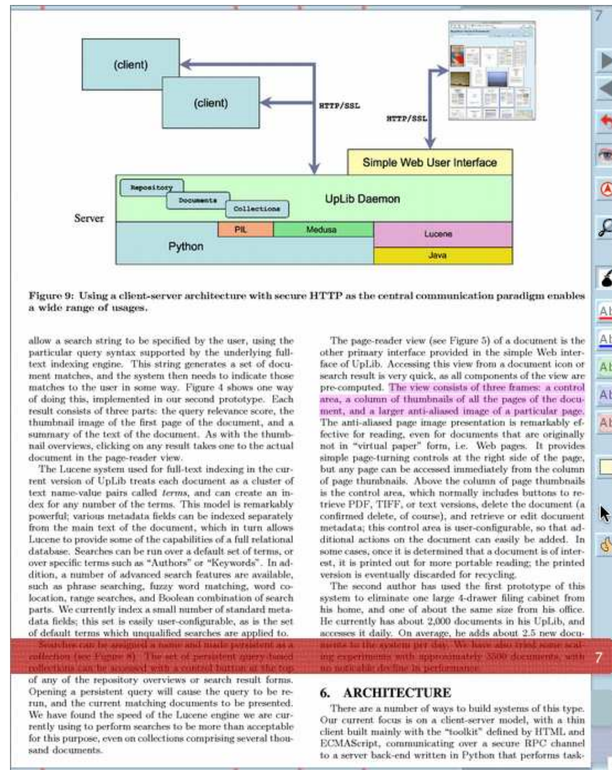


Figure 9: Using a client-server architecture with secure HTTP as the central communication paradigm enables a wide range of usages.

allow a search string to be specified by the user, using the particular query syntax supported by the underlying full-text indexing engine. This string generates a set of document matches, and the system then needs to indicate those matches to the user in some way. Figure 4 shows one way of doing this, implemented in our second prototype. Each result consists of three parts: the query relevance score, the thumbnail image of the first page of the document, and a summary of the text of the document. As with the thumbnail overviews, clicking on any result takes one to the actual document in the page-reader view.

The Lucene system used for full-text indexing in the current version of UpLib treats each document as a cluster of text name-value pairs called *terms*, and can create an index for any number of the terms. This model is remarkably powerful; various metadata fields can be indexed separately from the main text of the document, which in turn allows Lucene to provide some of the capabilities of a full relational database. Searches can be run over a default set of terms, or over specific terms such as "Authors" or "Keywords". In addition, a number of advanced search features are available, such as phrase searching, fuzzy word matching, word co-location, range searches, and Boolean combination of search parts. We currently index a small number of standard metadata fields; this set is easily user-configurable, as is the set of default terms which unqualified searches are applied to.

Searches can be assigned a name and made persistent as a collection (see Figure 8). The set of persistent query-based collections can be accessed with a control button at the top of any of the repository overviews or search result forms. Opening a persistent query will cause the query to be re-run, and the current matching documents to be presented. We have found the speed of the Lucene engine we are currently using to perform searches to be more than acceptable for this purpose, even on collections comprising several thousand documents.

The page-reader view (see Figure 5) of a document is the other primary interface provided in the simple Web interface of UpLib. Accessing this view from a document icon or search result is very quick, as all components of the view are pre-computed. The view consists of three frames: a control area, a column of thumbnails of all the pages of the document, and a larger anti-aliased image of a particular page. The anti-aliased page image presentation is remarkably effective for reading, even for documents that are originally not in "virtual paper" form, i.e. Web pages. It provides simple page-turning controls at the right side of the page, but any page can be accessed immediately from the column of page thumbnails. Above the column of page thumbnails is the control area, which normally includes buttons to retrieve PDF, TIFF, or text versions, delete the document (a confirmed delete, of course), and retrieve or edit document metadata; this control area is user-configurable, so that additional actions on the document can easily be added. In some cases, once it is determined that a document is of interest, it is printed out for more portable reading; the printed version is eventually discarded for recycling.

The second author has used the first prototype of this system to eliminate one large 4-drawer filing cabinet from his home, and one of about the same size from his office. He currently has about 2,000 documents in his UpLib, and accesses it daily. On average, he adds about 2.5 new documents to the system per day. We have also tried some scaling experiments with approximately 3500 documents, with no noticeable decline in performance.

6. ARCHITECTURE

There are a number of ways to build systems of this type. Our current focus is on a client-server model, with a thin client built mainly with the "toolkit" defined by HTML and ECMAScript, communicating over a secure RPC channel to a server back-end written in Python that performs task-

Fig. 1. A document open in an UpLib widget. The user has turned to page 7 by using a bookmark, and selected a region of text.

text of the document. Fourth, it extends some of the other page-turning mechanisms to support flipping back and forth between pages, something we refer to as *page twiddling*.

Turning pages. To turn from one page to the following page, a user can click on the "following page" button, an arrow pointing to the right, in the toolbar. Similarly, the "preceding page" button will turn to the preceding page in the document. The user can also left-click on the right side of the page to advance to the following page, or left-click on the left side of the page to move to the preceding page. With a keyboard, the arrow keys can be used, as can the "Page Up" (preceding page) or "Page Down" (following page) keys. Emacs keybindings are also supported; "Control-V" turns to the following page, "Meta-V" or "Control-P" turns to the preceding page.

When the reader turns from one page to another, a page-turn animation can be drawn for the reader. This consists of a vertical blue bar that moves smoothly across the widget either from right to left on a "following page" turn, or from

Fig. 2. Two-page viewing mode is useful for documents where the page design is spread across both pages.

left to right on a "preceding page" turn. On one side of the bar, the old page is drawn, horizontally scaled to fit in the space. On the other side, the new page is drawn, also horizontally scaled. The bar moves across the widget in a specified page-turn animation time, typically 300 to 400 milliseconds, specified by the application. Using a page-turn animation time of 0 effectively disables page-turn animations. Our experience to date, though, is that readers like to have the animations, probably to establish visually the change in context associated with a page-turn.

Non-sequential navigation. The ReadUp widget also provides several mechanisms for document navigation, which can be thought of as occasions when non-sequential page-turns are desired. At the top and bottom of the page image is an optional narrow horizontal sub-widget called a *page-edge*. The page-edge provides context to the user, visual feedback on where they are in the document. It can also be used for direct access to another part of the document. Clicking on a location in either page-edge turns to the page at a corresponding proportional location in the document.

Another mechanism provided for navigation is a bookmark system. Three bookmark "ribbons" are provided for each document, in three different colors. They are drawn as semi-transparent textured strips that protrude as tabs into the side toolbar, intended to suggest to the reader a thin silk ribbon bound into the spine of the "book". Each may be slid up and down the page, to the vertical position the reader finds most useful. A user "binds" a bookmark to a page by clicking on the tab of an unused bookmark when the document is open to the desired page. When a bookmark is bound – set to a particular page – its tab is drawn as a saturated end with a page number on it and a drop-shadow in the side toolbar (see figure 1). Unused bookmark tabs are drawn as faded, without

page numbers. When the reader clicks on the exposed tab of a bound bookmark, the document flips to the page the bookmark is set to. Clicking the tab of a bound bookmark while at that bookmark's page will unbind the bookmark, and it will disappear from that page; conceptually, it's pushed to the back of the book.

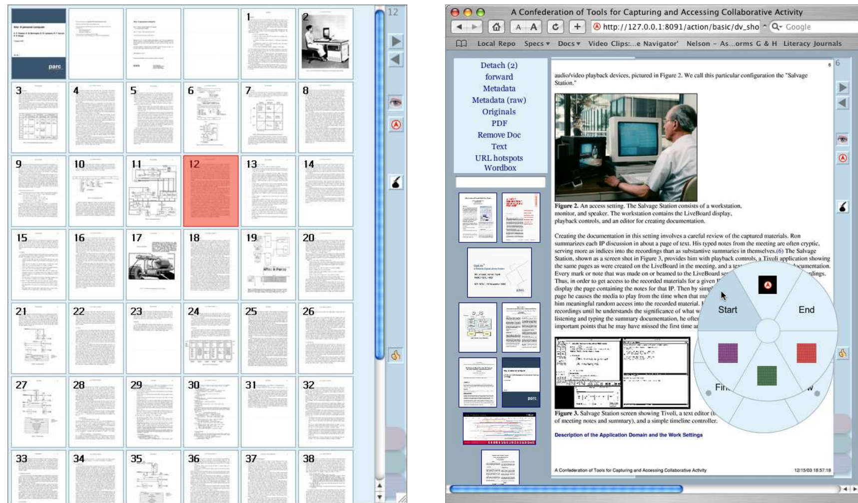
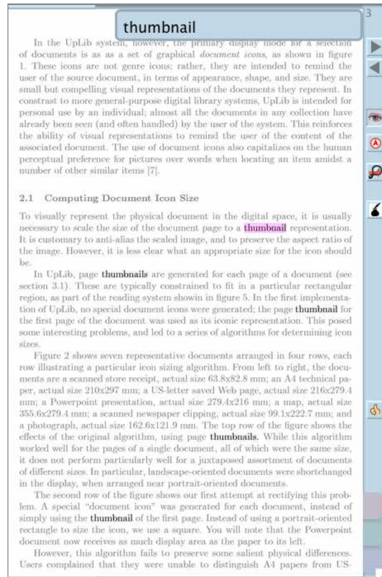


Fig. 3. (a) Page thumbnails shown in alternate view; the current page is highlighted. (b) A document displayed in a ReadUp applet in the UpLib Web interface; the reader is using the pie-menus to jump to the start of the document.

A third mechanism provided is document overview. The reader can switch to an alternate view of the document, shown in figure 3(a), either by pressing the "Alt" key on a keyboard, or by clicking on the small thumbnail button in the side toolbar. In this display, each page is shown as a small numbered thumbnail image. The current page is highlighted. The reader can switch to a different page simply by clicking on it, then switch back to full-page mode by either releasing the "Alt" key, double-clicking the page thumbnail, or clicking the thumbnail button again. This is often a useful way to find a particular page with distinctive graphics on it.

Text search. A major advantage of document navigation in virtual documents is the ability to search for text in the contents of the document. ReadUp provides a search mechanism modelled on the incremental text search mode of GNU Emacs [18]. When the user presses the search button in the side toolbar, or presses "Control-S" on a keyboard, the document goes into search mode. The page is desaturated, a search window is shown in the upper right-hand corner of the page image, and matching text is highlighted using "pop-out" perceptual

cues described in [19]. If text is selected before search mode is entered, that text becomes the search string; this makes it easy to select a word or phrase, then search for other occurrences of it in the document. Otherwise, the user enters text, one character at a time. As the search string is entered, the widget highlights all matches on the page. The "current" match is additionally highlighted with a pop-out color.



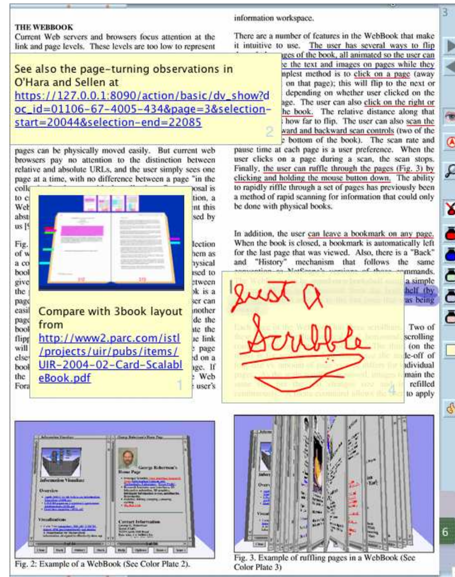


Fig. 4. (a) Text search in progress. All matches on the page are highlighted; current match is overdrawn with a pop-out color. (b) Annotations can be drawn directly on the page surface, or on note sheets, which also support text editing, pictures, and links. One of the links shown is to a part of another document in the reader's UpLib repository, the other to an "external" document on the Web.

A search in progress is shown in figure 4(a). The search string, "thumbnail", is shown in large letters in the search display. All strings matching that string are drawn with full contrast; the current match, in the abstract, is drawn with a violet pop-out highlight. The search button in the sidebar is overdrawn with a red right arrow to indicate that pressing it again will advance to the next match in the document, as will pressing "Control-S" on the keyboard while the search is active. If the next match is on a following page, the document will automatically be turned to that page.

The search window displays an indication of how many matches have been found so far, and whether or not the search has wrapped around from the end of the document back to the beginning. If the user advances to the end of the document, the search window turns red. If the current match is behind the search

window, the search window turns translucent so that the highlighted match can be seen through it. The search can be terminated either by pressing "Control-G" on the keyboard, or by clicking on the search window. When the search is terminated, the current match is turned into a selection.

In addition to this conventional text search, other less common modes of search can be provided by application logic. In the standalone UpLib reader, for example, clicking on the UpLib icon in the side toolbar brings up a window which allows a search for more documents in the UpLib repository. Similarly, two other standard searches are available through pie menu entries. One allows the reader to select a region of text, then converts that text passage into a search in the backing UpLib repository for other documents related to that text. The other search provides a similar service on Google, displaying search results in a Web browser window.

Page twiddling. It is common for readers to want to flip back and forth between two locations in a document. For instance, when reading a scholarly paper, readers often flip to the citations at the back of the paper as they occur in the text, then back to the text that they were reading, as they progress through the paper. Readers also often want to compare items on two different pages of a document, or flip back and forth between two adjacent pages to read a complete thought split across the two pages.

ReadUp provides two different mechanisms to support this. When clicking on a page, the actions of the right and left mouse buttons are reversed. This makes it very easy to flip back and forth between two adjacent pages without moving the mouse. Some pen users find it easier to just click the "following page" and "preceding page" buttons in the side toolbar.

The second mechanism is used with non-adjacent pages. Whenever a non-sequential page-turn is made by using a bookmark, or clicking in a page-edge, or changing pages in the overview, a *flipback* button is provided in the side toolbar. It can be seen in figure 1, just below the page-turn buttons on the toolbar. By pressing it, the document is turned back to the previous location. This change is itself a non-sequential page-turn, so the flipback button still appears, and if pressed will reverse the action of the the previous press.

A third somewhat experimental mechanism is also provided. If the reader is using a mouse with a mouse wheel, turning the mouse wheel will turn several pages in succession, the exact number depending on the amount the mouse wheel was turned. This allows the reader to flip back and forth through a number of pages fairly easily.

3.2 Annotation

Annotation mode, seen in figure 4(b), can be turned off and on by clicking on the image of an inkpot and quill pen in the side toolbar. When on, inkpots are visible on the side toolbar, past annotations are visible on the page, and new annotations can be made. When using a pen, the most convenient way

to annotate is to "dip" the pen in an inkpot, then write directly on the page. Structured highlighting or underlining of text passages can be accomplished by selecting the text region to be highlighted or underlined, then clicking on one of the annotation buttons in the side toolbar, which replace the inkpots when text is selected, as seen in figure 1. If a separate annotation or note is desired, clicking on the small yellow square button beneath the inkpots on the side toolbar will create a note sheet, which can be arbitrarily resized, moved around, and written on. Pages with annotations are indicated in the page-edges of ReadUp by small orange dots.

If a keyboard is available, the note sheets support a text editor, and text can be entered directly from the keyboard. In addition, note sheets support the inclusion of images and Web links. Links to selected areas of other UpLib documents can also be pasted into note sheets; when they are clicked on, the application can, for example, choose to open the other document in another window, with the selected region highlighted.

3.3 RSVP and ADH Modes

Two techniques for phrase-based reading are built into ReadUp, RSVP and ADH. Both segment the text of the document into short phrases using linguistic techniques. *Rapid serial visual presentation* (RSVP) [17] then presents each phrase in sequence in the center of the page image space at a user-controllable rate. *Animated dynamic highlighting* (ADH), on the other hand, presents each phrase in context, with the rest of the page text heavily desaturated, again at a user-controllable rate.

4 Implementation issues

Several issues were encountered during implementation of the the widget. Two of these merit separate discussion: management of the widget's graphics layout, and management of memory resources.

4.1 Graphics layout

Many of our reading situations involve a laptop, either with a trackpad or an external mouse, or a tablet-PC, used with a pen and virtual keyboard of some sort. Laptops have poor vertical resolution, so the toolbar was placed at the side to conserve vertical pixels. Tablet-PCs have few physical buttons to augment the pen (typically only two are usable), so buttons were placed on the toolbar to provide functionality already available to readers with a keyboard. In addition to these issues, the widget had to be embeddable in a larger application, which meant that it could not count on having access to menubars or other input systems outside the bounds of the widget. Conventional Java pop-up menus could be used, but pie menus are more usable by pen readers, so a hierarchical pie menu system derived from [8] was included (see figure 3(b)). Similarly, output

modes could not use space outside the bounds of the widget, so pop-up windows like that used for search (figure 4(b)) were allowed to carefully occlude part of the page image when necessary.

4.2 Resource management

Because ReadUp is a toolkit widget, a single application may have many instances of it active at any one time. For instance, a workspace for writing may have many reference documents open at the same time. Because ReadUp is designed to work with page-image digital documents, memory usage can be a problem. There are several memory-intensive data elements associated with the types of books ReadUp is designed for. Each page of the book is available as both a large clipped anti-aliased page image, and as a smaller page thumbnail. In addition, information on the text of the page, including bounding boxes, parts of speech, and other metadata, for each word, is usually available. Finally, annotations on the page, which may including drawings, text, and images, can occupy significant amounts of memory.

This is especially true for Java applications, which must work within the somewhat constrained memory model imposed by the standard Java virtual machine design. A large text, such as a reference work, can overwhelm the virtual machine if all of its data is loaded at the same time. To address this situation, ReadUp contains a dynamic caching mechanism.

Each type of data element (large page images, small page images, text metadata, and annotations) has a separate resource manager. Each resource manager implements the ResourceLoader interface, which provides a method called `getResource` to retrieve an instance of the resource, given the document, the page, and a selector which can be used to distinguish between different instances of the resource associated with the page, e.g. different annotations on the same page. Each manager maintains a cache of its resources. When another part of the ReadUp widget invokes `getResource` for a particular resource, the cache is examined, and if it contains the resource, it is returned. Otherwise (a situation known as a *cache miss*), the manager puts that resource on list of desired resources. This list is continually examined by a *resource loader thread* associated with the resource manager, which does nothing but bring desired resources into the cache. Once the request is available in the cache, the requesting part of ReadUp is informed with a callback, which also carries the requested object as a parameter.

The cache uses a *weak reference* [11] to point to the resource object. This is a type of reference which is not considered by the garbage collection algorithm, which means that if no other references to the resource exist, the garbage collector may reclaim the resource if it requires more memory. Resources which are in active use, such as a page image being rendered by the drawing subsystem of the ReadUp widget, have multiple "strong" references active, so they will not be garbage-collected. Unused resources, such as the images for pages not currently visible, may be reclaimed by the garbage collector if necessary. They will be automatically reloaded by the resource manager's resource loader thread if they

are later brought into use. Note that for resources freshly loaded in response to a cache miss, a strong reference to the resource is maintained by its use as a parameter to the callback method, which means that it may not be garbage-collected until the callback returns. This gives the callback code an opportunity to establish its own strong reference to the resource, if needed.

While ReadUp was explicitly developed for use with the UpLib system, it was also designed to be independent of that system. Resources can be loaded from arbitrary sources by providing appropriate resource loaders, which follow an abstract interface. Our standard loaders for use with UpLib load resources via network procedure calls, but other loaders can easily be constructed to load resources in other ways.

5 Current use of the ReadUp widget

The Web interface for UpLib has been redesigned to use ReadUp in a Java Plug-In applet. Figure 3(b) shows an example. The page thumbnails of the previous Web interface have been replaced with document icons for other documents in the repository, allowing the reader to easily flip back and forth between documents, and an UpLib search window has been added. All of the aforementioned deficiencies of the previous Web interface have been remedied.

The ReadUp widget is already being used in other applications such as the corpus browser described in [3], and the ReadUp application, which is a simple search-and-display wrapper around the widget. Because the widget handles all the details of supporting the reader in the reading process, these applications are able to concentrate on collection navigation and management. In addition, the consistency of reading operations across the various applications lowers the learning curve for new users.

6 Acknowledgements

The UpLib system itself is the result of joint work with Ashok Popat at PARC. Many of our colleagues at PARC have contributed generously to our work on this project, notably Lance Good, Jeff Breidenbach, Eric Bier, and Stuart Card.

References

1. A. Adler, A. Gujar, B. L. Harrison, K. O'Hara, and A. Sellen. A diary study of work-related reading: design implications for digital reading devices. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 241–248. ACM Press/Addison-Wesley Publishing Co., 1998.
2. M. J. Adler and C. V. Doren. *How to Read a Book*. Touchstone Books, revised edition, 1972.
3. E. Bier, L. Good, K. Popat, and A. Newberger. A document corpus browser for in-depth reading. In *JCDL '04: Proceedings of the Fourth ACM/IEEE Joint Conference on Digital Libraries*, pages 87–96, June 2004.

4. S. K. Card, L. Hong, J. D. Mackinlay, and E. H. Chi. 3book: a scalable 3d virtual book. In *Extended abstracts of the 2004 conference on Human factors and computing systems (CHI)*, pages 1095–1098. ACM Press, 2004.
5. D. Chaiken, M. Hayter, J. Kistler, and D. Redell. The virtual book. Technical Report 157, Digital Equipment Corporation Systems Research Center, November 1998.
6. Y.-C. Chu, D. Bainbridge, M. Jones, and I. H. Witten. Realistic books: a bizarre homage to an obsolete medium? In *JCDL '04: Proceedings of the 4th ACM/IEEE-CS joint conference on Digital libraries*, pages 78–86. ACM Press, 2004. http://www.nzdl.org/html/open_the_book/.
7. A. Dillon. Reading from paper versus screens: A critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, October 1992.
8. J. Hong. Java pie menus. World Wide Web, 2002. http://www.cs.berkeley.edu/ jasonh/download/software/piemenu/.
9. W. C. Janssen. Document icons and page thumbnails: Issues in construction of document thumbnails for page-image digital libraries. In *ECDL 2004: Proceedings of the Eighth European Conference on Digital Libraries*, pages 111–121, 2004.
10. W. C. Janssen and K. Popat. UpLib: A universal personal digital library system. In *DocEng 2003: Proceedings of the ACM symposium on Document Engineering*, pages 234–242. ACM Press, November 2003.
11. R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.
12. C. C. Marshall. Annotation: from paper books to the digital library. In *DL '97: Proceedings of the second ACM international conference on Digital libraries*, pages 131–140. ACM Press, 1997.
13. K. O'Hara and A. Sellen. A comparison of reading paper and on-line documents. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 335–342. ACM Press, 1997.
14. K. O'Hara, F. Smith, W. Newman, and A. Sellen. Student readers' use of library documents: implications for library technologies. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 233–240. ACM Press/Addison-Wesley Publishing Co., 1998.
15. T. A. Phelps and R. Wilensky. The Multivalent browser: a platform for new ideas. In *DocEng '01: Proceedings of the ACM Symposium on Document Engineering*, pages 58–67, Atlanta, Georgia, 2001. ACM.
16. B. N. Schilit, G. Golovchinsky, and M. N. Price. Beyond paper: supporting active reading with free form digital ink annotations. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM Press/Addison-Wesley Publishing Co., 1998.
17. K. Sicheritz. Applying the rapid serial presentation technique to personal digital assistants, 2000. Master's Thesis, Department of Linguistics, Uppsala Universiity.
18. R. M. Stallman. *GNU EMACS Manual*. Free Software Foundation, 2000.
19. B. Suh, A. Woodruff, R. Rosenholtz, and A. Glass. Popout prism: adding perceptual principles to overview+detail document interfaces. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 251–258. ACM Press, 2002.
20. Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.0: API Specification*, 2002.