Making UpLib Useful: Personal Document Engineering

William C. Janssen, Jeff Breidenbach, Lance Good, Ashok Popat

4 July 2005

TR−05−5

parc℠
Palo Alto Research Center

For more information on PARC, please visit our Web site at http://www.parc.com/.
Our address is:

Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA

You can contact us via telephone at 650−812−4000.
You can also send e−mail to info@parc.com; it will be forwarded appropriately.

Making UpLib Useful: Personal Document Engineering

William C. Janssen, Jeff Breidenbach, Lance Good, Ashok Popat
Palo Alto Research Center, Inc.
3333 Coyote Hill Road
Palo Alto, CA 94304

email: janssen@parc.xerox.com

4 July 2005

Abstract

Any new system must provide significant advantages to users for them to adopt it over their existing practices. In this paper, we discuss changes made over the last two years of use of the UpLib personal digital library system, to provide those advantages in the realm of personal document management. These changes are concentrated in the document acquisition phase, where document analysis is performed and databases of document information are prepared. However, some changes have been made in the areas of collection management, and general document usage, primarily to provide better interfaces to the improved document projections.

1 Introduction

The UpLib personal digital library system [21] provides a secure long-term storage and retrieval system for a wide variety of personal documents such as papers, photos, music, bills, books, and email. It is suitable for collections comprising tens of thousands of documents, and provided for ease of document entry and access as well as high levels of security and privacy. It is highly extensible through user scripting. It aspires to Bush's idea of the *memex*, "a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility" [5].

UpLib is implemented as a service, the "guardian angel", which manages a directory containing all of the user's archived documents. Requests for various document services may be made to the server either through a secure Web browser interface, or via encrypted RPC from various client programs. These services may include adding a new document to the repository, finding some set of documents matching specific criteria, or fetching a document to read, either via the Web, or in a separate display window. New capabilities may be added individually to any repository via the built-in extension mechansim. UpLib is thus designed to be useful as a platform for research into digital libraries and computer-augmented reading.

But for most users, holding and serving documents is only half the story. A personal library must also solve personal problems in document engineering, problems like dealing with paper documents, cleaning poorly scanned documents, understanding new document formats, or saving web pages and mail attachments. There is also the much more complex problem of actually using – typically reading – the archived documents.

In this paper, we discuss a number of changes we have made to the original UpLib system to support discovered requirements for personal digital libraries. UpLib can be factored into three phases: *document acquisition*, when a document is captured and analyzed by UpLib, and various databases are updated; *document management*, where documents in the system are examined through browsing and search, and metadata is updated; *document use*, where a user interacts with a stored document for some external purpose, such as reading or use of the document as source material in preparation of a talk or paper. Most of our changes are concentrated in the document acquisition phase; however, some significant changes were needed for both the management and use parts of the system.

2 Document Acquisition

Many of the changes we needed to make were to make capture of documents easier, and our automated analysis of the saved documents more thorough. We added two new input clients, to handle mail attachments, web pages, and paper documents with low user effort. We also made a number of changes to the automated analysis engines to give us better page images, word boxes for document pages containing text, support for larger documents, and user-extensible document format parsing.

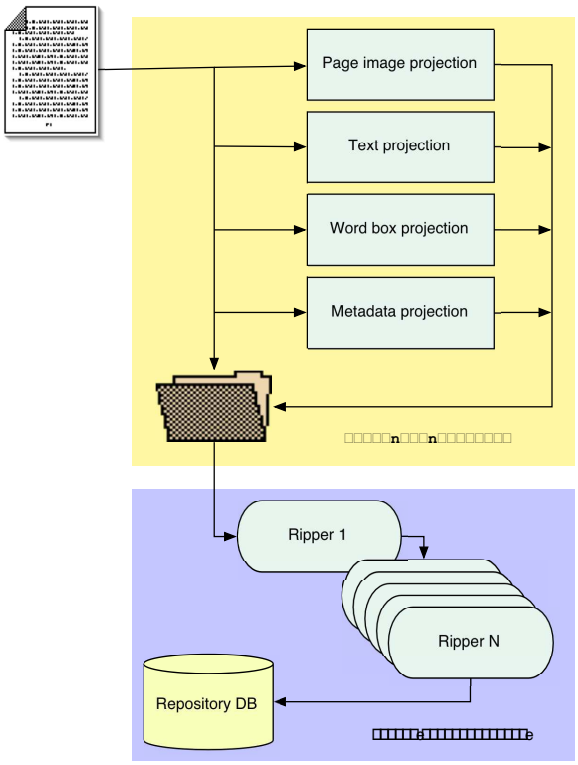

Figure 1: Adding a document to an UpLib repository. Projections are formed in the `uplib-add-document` client, merged with the document original, and sent to the server, where document analysis "rippers" are run.

2.1 Low-effort Capture Clients

As Brown et. al. note in [4], previous work on document capture doesn't say much about how or why people capture information for personal use. With recent rapid changes in hard-disk capacity and digital photo capability, capture techniques have changed equally rapidly, moving decisively into the digital domain. In addition to the now familiar use of email [12], people bookmark Web pages, archive chat sessions, send each other cell-phone snapshots. Recent work by Marshall and Bly [22] suggests that even clipping of newspaper and magazine articles is now moving online.

Another characteristic of this trend is the use of low-effort, low-understanding user interfaces. The user now expects to have everything happen automatically at the push of a button (and rightly so, in our opinion). Efforts such as [16] have exploited the push of actual hardware buttons by modifying digital copiers to automatically capture paper documents, but have not addressed low-impact intentional capture of electronic documents. Other systems [9, 13, 2] take advantage of side-effect caching by programs such as Web browsers and mail readers, providing easier search access to whatever happens to be on the user's disk, but with little discrimination and apparently no long-term archival support. Research platforms such as Haystack [17] also support automatic collection of things such as email or Web pages by interposing proxy agents between the user and the document source, and add some discrimination by using the history of user queries to pre-filter the collection.

In the original version of UpLib, we provided a command-line program, `uplib-add-document`, which, given a file or files, performed all of the client-side processing necessary for document acquisition, and sent the document to the UpLib server. The server assigns the new document an identifier and folder, and make it available for subsequent use. `uplib-add-document` supports command-line options which allow the user to provide metadata, or control certain aspects of document processing, such as whether to perform OCR to retrieve text from the document's page images. However, in looking at what users wanted to archive with UpLib, we found that many, perhaps most, documents were not files on a filesystem, but rather either (1) pages of paper, (2) mail attachments, (3) images on a Web page, or (4) Web pages. To support these usages, we introduced two new mechanisms for adding documents to an UpLib repository.

2.1.1 The UpLib scan service

The first capture mechanism is designed for paper documents. It uses Flowport [28], a Xerox system which allows a user to control document distribution by interpreting marks made on a cover sheet when the document is run through a Xerox digital scanner. With UpLib, the user checks one or more boxes on their cover sheet (see figure 2) to indicate which categories the document is to be placed in, then places the cover sheet on top of the document to be scanned, and runs both through the scanner. The scanned document automatically shows up in the user's UpLib repository.

This service is implemented by having the Flowport server OCR the scan job, convert it to a PDF file, and send the file, along with any metadata specified on the cover sheet, to a second server which interprets the metadata from the cover sheet, and uses `uplib-add-document` to send the rest of the scan job to the appropriate repository. In addition, the service provides OCR'd text of each page for use by the rest of the system.

The UpLib scan service is available with any Flowport-enabled scanner at our site. These scanners (which are also printers, copiers, and fax machines) are located every hundred feet or so on each floor of our building.

2.1.2 The UpLib Portal

The second mechanism, the UpLib Portal, is a drag-and-drop interface. It allows the user to drag a mail attachment from an open email message, or a link or image or entire web page from a web browser, and drop it on a small icon on the user's desktop. When the user releases the dragged document, a submission metadata form pops up, to allow the

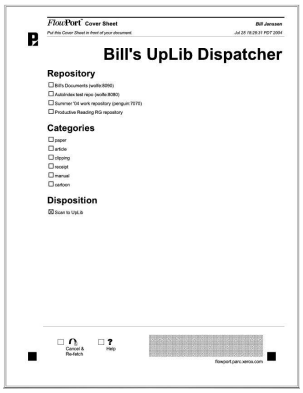

Figure 2: A Flowport cover sheet for the UpLib Scan Service.

user to optionally enter metadata about the document (figure 3a). The user may choose to either add metadata, or ignore it; in either case, pressing on the "Submit" button causes the portal to begin processing the document by submitting it to `uplib-add-document` in a subprocess. Thus the minimum user effort needed to enter a visible document is a drag plus a confirming click. The user can also shift-click on the Portal to open a file selector window, to permit the entry of a specific file.

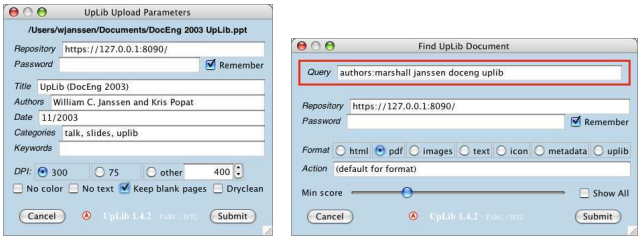

Figure 3: (a) The Portal submission window. (b) The Portal query window.

The Portal also serves as a retrieval system. Clicking on the Portal icon brings up a query form (figure 3b). The user can type an UpLib query and press "Enter"; results are (by default) opened directly in a new browser window. In addition to supporting search-based retrieval, this interface allows the user to fetch various versions of the document (such as the document icon or a PDF version); to adjust the search sensitivity via a slider; and to fetch either only the highest-scoring match, or all matches above the sensitivity threshold. When the user submits the query, the Portal invokes the command-line program `uplib-get-document` in a subprocess to perform the retrieval.

2.2 Document Analysis

When a document is added to an UpLib repository, it undergoes a process called *projection*, in which projections of the document into various simple spaces are performed. Our initial system extracted three projections of the document: into page images in a multi-page TIFF file, into metadata name-value pairs in a text file, and into text in a text file. The use of multi-page TIFF proved unwieldy for large documents, and we have changed our system to instead use a directory containing numbered PNG image files, one for each page. We have also added a new projection, word boxes. Our initial system supported a fixed set of document formats; we have added a mechanism which supports a flexible and user-extensible set of document formats. We have also added a new pre-projection service, called *document dry-cleaning*.

2.2.1 Word boxes

To support user manipulation of text in document views, we added a new projection to our system which we call *word boxes*. This is a description of the bounding box of each text word on each page image of the document, arranged in reading order. To create this projection, we use two techniques. The first is used for document formats which can be coerced into PDF, such as Microsoft Word or HTML. We have developed an extension of the program `pdftotext`, from the `Xpdf` toolkit [23], which when run against a PDF file provides a list of the words encountered and their bounding boxes, along with potentially interesting information such as whether the font used for that word is bold or italic. This list is converted into a binary format and stored along with the other projections of the document.

In some cases, the PDF form of the document cannot be interpreted correctly by `pdftotext`. We evaluate the output of `pdftotext` by comparing it to a statistical model of word usage in the dominant language of the document, using a tool called `scoretext` (see below). If it is too dissimilar, we discard the output of `pdftotext`, and turn to our fallback mechanism, which is also used if the document cannot reasonably be coerced into PDF format. This relies on using Inxight's TextBridge OCR system to OCR the document, returning results in the XDOC format [27]. We interpret the XDOC results, converting from them to the binary format used in our projection. We have also developed a similar pathway from ABBYY OCR output formats.

When the document is uploaded to the UpLib guardian angel, a ripper processes the word boxes file into smaller files, each containing the word box information and text of a single page. The word box information is transformed to reflect the scaling and translation of the reduced-DPI page images formed as part of the normalization step of the UpLib document incorporation pipelined described in [21].

2.2.2 Evaluating text projections with `scoretext`

For any given document, generation of the text projection can usually be carried out in several different ways. For instance,

if the document is a PDF file, one could run any of several programs designed to extract the text layer from the file if present. If the text layer is not present, as might be the case when the PDF was generated by scanning a paper document without OCRing, an OCR step can be performed to generate the textual version. Moreover, there may be a choice between several available OCR programs. If the scanned document happens to be clean, has a simple layout, and uses a single well-known font, then an open-source program like *ocrad* might perform sufficiently well. On the other hand, if the document has complex layout or has other difficult-to-OCR characteristics, then use of a commercial OCR system may be required. In general, the different means of generating the text projection will result in varying levels of accuracy and will also present different costs in terms of time and possibly licensing fees. It is desirable therefore to be able to choose the method of conversion according to the required accuracy and estimated cost. We assume that reliable estimates of the time and licensing costs are available, and focus on automated estimation of the accuracy of the projection with a program called `scoretext`. The general approach is to compute the linguistic validity of the text with respect to a precomputed language model.

In its full generality, a probabilistic language model would specify a probability distribution over all finite sequences over a given alphabet, for instance UTF-8. For simplicity we restrict the language model to be factorable as a sequence of probability distributions over individual characters, each conditioned on a subset of preceding characters. Let the alphabet be $\mathcal{A}$, and let $v_1, \ldots, v_n$ denote a string with $v_i \in \mathcal{A}, i = 1, \ldots, n$. Let τ be a termination symbol, and let $\mathcal{A}' = \mathcal{A} \cup \{\tau\}$. We view strings as having been formed by the following process. Characters are generated sequentially according to a sequence of conditional probability distributions

$$p_i(v_i|v_1, \ldots, v_{i-1}) = p_i(v_i|\phi_i(v_1, \ldots, v_{i-1})) \qquad (1)$$

where $v_i \in \mathcal{A}'$, $v_1, \ldots, v_{i-1} \in \mathcal{A}$, $i = 1, 2, \ldots$, and the function $\phi_i(v_1, \ldots, v_{i-1})$ maps contexts into equivalence classes. The string terminates when the symbol τ is generated.

For simplicity, we remove the dependence of p in (1) on i, and restrict $\phi_i(v_1, \ldots, v_{i-1})$ to be of the form

$$\phi_i(v_1, \ldots, v_{i-1}) = \begin{cases} (v_1, \ldots, v_{i-1}) & \text{if } i \leq N \\ (v_{i-N+1}, \ldots, v_{i-1}) & \text{otherwise} \end{cases} \qquad (2)$$

for a fixed small integer N. With these restrictions, (1) is referred to as a *character N-gram language model*, hereinafter referred to simply as an N-gram model:

$$p_i(v_i|v_1, \ldots, v_{i-1}) = p(v_i|v_{i-N_i+1}, \ldots, v_{i-1}) \qquad (3)$$

where $N_i = \min\{i, N\}$. Such N-grams are simple and effective in capturing important statistical regularity in natural language strings. We therefore adopt their use here. To illustrate the ability of the N-gram to model English, we exhibit a pseudorandom string that was obtained by sequentially sampling from a 5-gram model trained on the Brown corpus [14]:

```
And fine alone other Itality and
agains she tumor his result, from
of Brannot faculous shall
precentative inter at lear time to
much a relanguages proposal?  Baer
fall over Open-megaw of most used
agreen during represearchbishes of
chlor] But the first position a
little rear intenant year-olds to
And also if we should beef
childing face graduates
```

Once the language model $p_i(v_i|v_1, \ldots, v_{i-1})$ has been trained on example data from the target language, it is used to assess the quality of the output of a text-conversion process by computing the per-letter coding cost (in bits) of the sequence with respect to the model. `scoretext` prints a score directly related to the average number of bits per letter that would be emitted by an ideal compressor when using the N-gram model, on the subject text. At the beginning of the document some of the conditioning characters specified by the N-gram model will be unavailable, corresponding to character locations before the first character in the converted text. These unavailable characters are taken by `scoretext` to be space characters. The score reported by `scoretext` is then used to decide whether or not the text file contains linguistically valid text by comparing it with a user-specified threshold. In principle, the score could be used to choose among several competing text-conversion results; currently it is used to simply accept or reject each text-conversion attempt considered in a predetermined sequence.

2.2.3 The format parser framework

The world delights in inventing new formats in which to store documents, and users sometimes seem to have a perverse talent for seeking out those formats and using them. To cope with this, we have defined an extensible framework for handling new document formats. Our format parser is defined in Python, a dynamically-typed language with automatic compilation of modified sources. Each document format (PDF, Microsoft Word, TIFF, etc.) is handled by a subclass of the base class `DocumentParser`. The input analyzer uses introspection to construct a list of all available parser classes, and calls the `myformat` class method of each class, passing the document as an argument, until one of the parser classes indicates that it should handle the document's format. The order in which these tests are performed may be partially controlled by the use of `BEFORE` and `AFTER` clauses in each class, which can specify a partial order with respect to other classes.

When the appropriate class is identified, an instance of the discovered class is instantiated, bound to the input document,

and the `process` method of the instance is invoked, which in turn invokes other methods for constructing the text, page-image, and word-box projections, or extracting metadata elements. Each of these methods can be overridden in a format-specific subclass to provide special processing. For example, the JPEG parser overrides the metadata-extraction method to handle embedded EXIF data in the image.

Users can easily extend this framework for new document formats by defining new parser classes (which can just be subclasses of existing parser classes) in a file and making the file known to the system by setting an appropriate user configuration parameter. Since Python automatically compiles modified source files, there is no need for the user to perform explicit compilation or linking. Parser classes defined in user files are integrated into the list of parsers formed by the input analyzer. Since each user file will be reloaded on each invocation of the `uplib-add-document` command, the new parser class is easily debugged without re-starting or re-installing the UpLib system. To further aid in debugging, we have added the `--noupload` switch to `uplib-add-document`, which performs the full projection calculations, but instead of uploading the resultant files to the UpLib guardian angel, leaves them in a temporary directory.

2.2.4 Document dry-cleaning

Scanned documents frequently contain some visual noise which users would rather not see. To handle this, we added an optional "dry-cleaning" operation on input images, which consists of two stages. The first stage removes pepper (small noise flecks) by using standard morphological operations on a scaled-down binarized form of the image. The second stage estimates the skew of the page using two different skew detection techniques. If a skew angle is detected with sufficient confidence, a high-speed rotation operation is performed which moves around all the pixels of the page image; no interpolation is performed, though an attempt is made to avoid long horizontal shear lines.

Dry-cleaning is only performed on the page image projection of the document. If original page images are present, they are not modified in any way. The page image projection is later used to create anti-aliased reduced-scale versions of the page images, which are then used in the page-reading clients. Thus later usage is almost always of the dry-cleaned version of the document, unless the user specifically requests the original version from the UpLib repository.

2.2.5 Dealing with Web pages

Capturing Web pages becomes an interesting challenge. Pages in relatively well-behaved single-file formats such as PDF or Microsoft Word are simply copied locally; the MIME type of the page is used to select an appropriate format parser class. However, HTML is a poorly behaved document format, as it

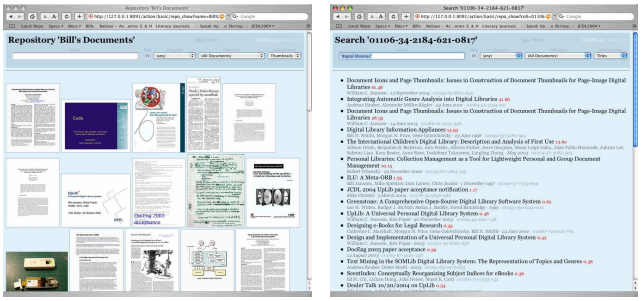

Figure 4: (a) The default view of a repository. The display shows thumbnails for the most-recently-used (or added) documents. (b) A textual display. We show the title, authors, date, and document ID. Each author's name is a link to a search for other papers by that author. Parts of the date are links to searches for other documents with that date.

can have dangling references when simply copied. To address this, we restrict our support to two kinds of HTML documents: the Mozilla "Web Page Complete" format, and an HTML page on a remote Web server. We use `HTMLDOC` [1] both as a Web spider for remote pages, and to convert the HTML into PDF for projection processing.

Users expect to be able to capture any page that they can see with their Web browser. To accomplish this, our HTML format parsers can read browser cookie files, so that they can ask for the remote page with all the capabilities that the browser would have. In addition, we modified the remote Web page format parser and `HTMLDOC` to send a "Referer" header when fetching a page, as some web sites require it.

3 Managing Documents

Once a digital library contains thousands of documents, the reader must have ways of browsing and searching this corpus, to find the specific documents they are interested in working with. The UpLib Web interface provides both visual and textual displays of subsets of the document space, and allows searches with a powerful search engine. Results are displayed in a highly visual form, allowing users to exploit their perceptual abilities to locate the correct document from a small set of search results. Alternate display modes can be used for particular tasks.

3.1 Viewing the Repository

Document icons are generated automatically for any document placed in an UpLib repository (see [21] for details). Figure 4a shows that these icons are often visually distinctive; figure 5b shows that this is not always true. Colored labels can now be added to nondescript document icons to bolster their visual identity; one is shown in figure 4a. The size of the icon is related to the size of the original document according to

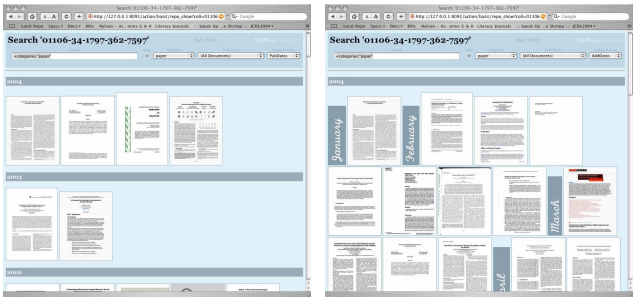

Figure 5: Date-oriented displays of a collection. (a) On the left, the collection is sorted by publication date. Labelled horizontal stripes divide years. (b) On the right, the collection is sorted by date of addition to the user's library. Note that month tombstones divide the thumbnails, when more than 5 documents occur in one year.

the "log-area" algorithm described in an earlier paper on this work, [19], and gives the user an additional visual cue as to the document identity.

The default top-level display of an UpLib repository shows document icons for each of the *N* most recently used documents. An example is shown in figure 4a, where we can see 15 of the most recently used or added documents without scrolling the page. This display can be varied from thumbnail to textual (figure 4b), or a combination view which shows the thumbnail along with the title, abstract or comment, authors, and date. In our original system, all three of these views showed the most recently used documents in the repository.

In our current system, each of these three views can also be varied to show the results sorted by either date of document publication (figure 5a), or by date of addition to the repository (figure 5b), two views inspired by the work Bier et. al. describe in [3]. When either of the date-based views is used, the most recent year is displayed at the top of the window, and documents are ordered from oldest to newest within that year. If a year includes more than five documents (this number is user-configurable), month markers are shown between documents published or added in different months.

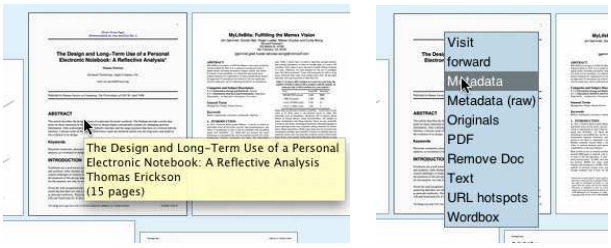

Figure 6: (a) Each icon has a tool-tip with title, author, and page count. (b) Each icon has a pop-up menu to invoke actions on that document; here the user has dragged down to the "Metadata" action, which opens a metadata editor window.

Each icon is "tool-tipped" with the title and author of the document, as seen in figure 6. Each icon also serves as the anchor for a CSS/Javascript pop-up menu of actions that can be performed on the document. Simply clicking on the icon will invoke the default action, "Visit", which opens the document in a reader. Users can add actions to this menu in their configuration files; UpLib extensions [18] can also add actions.

3.2 Categories and Collections

Like Placeless [10] and MyLifeBits [15], UpLib does not attempt to associate documents with specific locations on a filesystem. While this has advantages from a document management point of view, users expect a document to have some sort of location where they can find it again when they need it. To provide this sense of "place", a digital library system needs to offer other kinds of locations in which users can expect to find particular documents. UpLib provides two such concepts, the *category* and the *collection*. Both have been modified slightly from the original design.

Documents can be associated with particular category names. Our original system just used a flat namespace of categories; in our most recent version, this has been changed to allow a hierarchical naming system. These categories form a hierarchical personal "concept map" for each repository. Documents can appear in any number of categories, thus having multiple "places" where they can be found. New categories can be created simply by associating the category name with one or more documents. One of the principal ways of finding documents in UpLib is to retrieve all the documents in a category, and then use either the thumbnail or textual view of that subset to find the desired document or documents.

Collections in the original system were simply named queries; when the user "opened" the collection, the query was run, and the results presented. In our current system, we have moved to the slightly more sophisticated form of named query used in Presto [11]; a query is run against the document corpus, and the results of the query are merged with explicit inclusion or exclusion of specific documents to form the collection's contents.

3.3 Searching for Documents

Each document entered into an UpLib repository is full-text indexed with an application that uses the Lucene library [26] for indexing and search. Because Lucene can index named fields for each document, metadata fields such as title, authors, or publication date can be indexed along with the text of the document, and used as specific search terms limited to those metadata fields. However, searches on plain text terms will also work; those terms are searched for in the content of the document, and in various appropriate metadata fields. The exact set of metadata fields searched over can be configured by the user.

We have made several modifications to the original search system. In the original, we used the Lucene query parser in its default mode of combining multiple search terms. For instance, the search "authors:janssen ebooks" would find all documents either authored by Janssen, or containing the word "ebooks". To find all documents authored by Janssen and containing the word "ebooks", the query would have to specified as "authors:janssen AND ebooks". Users found this confusing, mainly because Web search engines default to ANDing of search terms. We developed a modified multi-field query parser which would support this usage, and changed the default.

Another issue poorly supported by our original system was the exclusionary search, such as "show me all documents NOT about ebooks". The default Lucene query system did not support such queries, mainly because there is no way to indicate "all documents". To address this, we added a pseudo-category called "_(any)_", to which all documents are added. We then modified the query parser so that queries which consist solely of exclusionary elements are automatically rewritten to include "categories:_(any)_" as an inclusionary clause. Thus, the query "-ebooks" is rewritten as "categories:_(any)_ AND -ebooks", which provides the effect desired by the user.

The results of a search can be displayed in the various formats discussed earlier. Figure 5a shows a typical date-published icon listing: the document icon is displayed; the search score is available in the tooltip for the icon. Clicking on the icon opens the document in the document reader discussed below. The results of a search can also be saved as a named *collection*. Saved collections can be easily accessed through a pull-down menu at the top of the UpLib repository Web view.

3.4 Replacing Document Content

We have found a number of situations in which the user would like to replace the content of an existing document with a new version of that content. For example, in one application, a technical paper would be parsed to find citations from that paper to other papers [3]. The application created "ghost" documents in the UpLib repository for each of the cited papers, which contained only some metadata, but no real content. This allowed them to get document identifiers for that document from the repository, which could be used as valid UpLib document identifiers in the application. The application would then at some point in the paper encounter a full copy of the cited work, and wish to replace the ghost entry with the valid copy. In another case, the user wanted to continually update the stored document to the latest version as it was revised over a period of months, but without creating multiple copies of the document and its metadata in the repository.

As reported in [21], when a document is added to an UpLib repository, it is processed by a series of document analysis engines, which we call *rippers*. This chain of rippers can be extended or arbitrarily modified through installable extensions to UpLib which can be downloaded on a per-repository basis from a central extension library, as described in [18]. We decided to support replacement of content by adding a powerful general mechanism which allows any ripper to abort the incorporation of a new document into the repository. This is done by raising a specific exception, `AbortDocumentIncorporation`, from the code of a ripper.

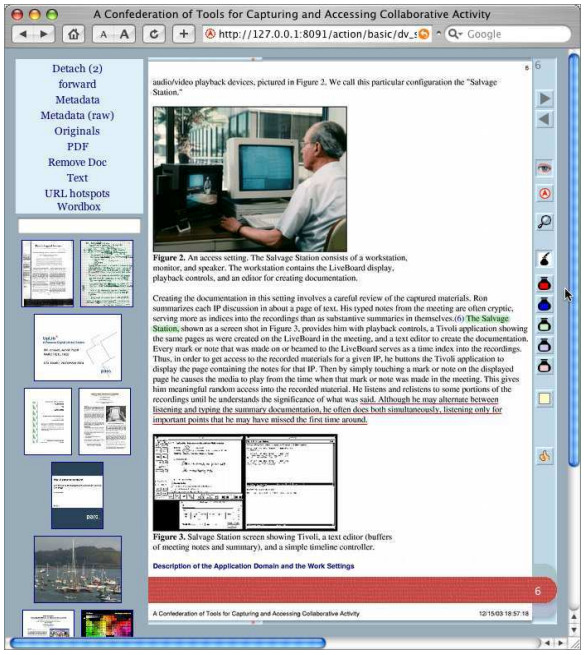

Figure 7: The current UpLib Web interface. Operations that can be performed on the document are shown in the upper left, along with a search box to find other document. Other recently used documents are shown in the lower left; clicking on one of them switches to a display of that document.

This allows the implementation of pseudo-rippers which can accomplish the replacement. For example, a ripper to handle replacement of ghost documents with true data would examine the metadata of a new candidate document. If it encountered a specific metadata tag, the value of which would be a valid document id, it would copy the new document's data into the folder of the existing document, start a re-indexing of the modified document, and abort the incorporation of the new document, now determined to be simply replacement content for an existing document.

4 Reading Documents

In the original system reported in [21], documents were read in a Web browser, using a page-reader view. A document page image was presented on the right side of a Web page; docu-

ment page thumbnails were available in a column to the left of the page display. Clicking on the right side of the page would turn to the next page; clicking on the left side, the previous page. Explicit next-page and last-page buttons were also visible in a control strip to the right of the document page image.

This interface suffered from a number of deficiencies. Too few page thumbmails were visible along the left side to be really useful, and the column of thumbnails did not track the position of the user in the document. The successive large page images cluttered the browser history, and made it difficult to "get out of" the document. There was no way to find text passages in a text document. Perhaps the most commonly remarked problem was that there was no ability to annotate pages as they were being read. To address these problems, we developed a new reading system called "ReadUp". ReadUp is implemented as a Java Swing widget, and can be deployed in either standalone Java applications (clients for UpLib), or in applet form suitable for use in a browser page.

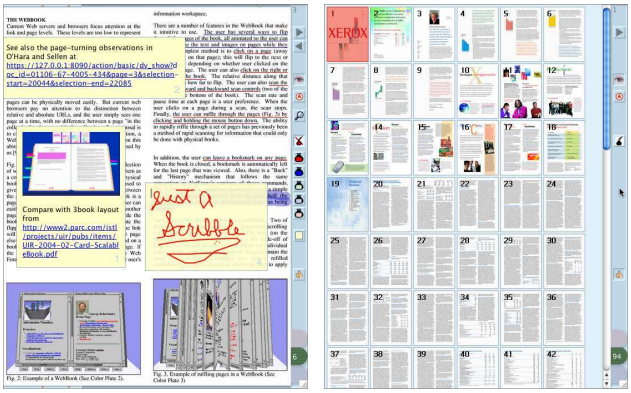

Figure 8: (a) An annotated page. (b) The overview mode, showing page thumbnails.

4.1 The ReadUp Widget

The ReadUp widget is implemented as a user-interface element for the Java Swing user-interface toolkit, along with a number of auxiliary classes that allow it to read from, and write to, an UpLib server. In this section, we briefly describe the design and operation of the widget; a full description is available as [20].

In layout, the widget mimics the earlier reading interface designed for the first version of UpLib: a page image is displayed, along with a sidebar which provides page-turning and other controls. An new visual feature is the presence of *page-edge* indicators/controls at the top and bottom of the page image, which provide the user with an indication of where they are in the document, and direct access to other parts of the document. Pages can be turned in various ways: by clicking on the left or right side of the page, by "riffling" throught the pages with the mouse wheel, by using bookmark "ribbons", etc. The reader can view all the pages of the document laid out as thumbnails (figure 8b), and jump through the document by clicking on those thumbnails. Page turns may be animated. The interface can be operated with either a mouse or a pen.

The XLibris system [25] may have been the first to support direct marking of the virtual paper sheets with a pen. In ReadUp, the user can also draw directly on the document pages with a variety of virtual pens and highlighters, or on separate note sheets, which also support a text editor with support for image pasting and hypertext links (see figure 8a). Incremental text search in the document is supported, as is selection of text and copying of it to the system clipboard, or to note sheets.

The design of ReadUp was heavily influenced by the O'Hara and Sellen study reported in [24], and by the prior PARC work on WebBook [7]. O'Hara and Sellen suggest three major requirements for future "paper-like" reading systems: recognize that annotation can be an integral part of reading and build support for it; quicker, more effortless navigation techniques; more flexibility and control in spatial layout. We believe that ReadUp's annotation and navigation facilities ably support the first two requirements, and its implementation as a widget, rather than as a standalone application, makes it easier for applications to achieve the third objective.

Behind the visible interface, ReadUp also provides some resource management algorithms to decrease coupling between interface client and UpLib server. Memory-intensive resources such as page images are kept in a two-level client side cache that is integrated with the Java virtual machine's garbage collector, so that unused resources can be reclaimed as necessary. Upon re-use of that resource, the reclaimed resource is either loaded from a local disk cache, or re-fetched from the UpLib repository, as needed.

Unlike systems such as Open-The-Book [8] and 3Book [6], ReadUp is implemented as a pure Java widget which does not need any non-standard libraries. This allows it to be used in systems which do not permit foreign libraries to be installed, such as Web applets. ReadUp has been integrated into the Web interface for UpLib by using applet technology. Figure 7 shows the current reading interface, using the ReadUp widget running in a Java Plug-in applet. In addition, the ReadUp technology is being used in several other UpLib clients, such as the work described in [3], and in a simple document retrieval application described in the next section.

4.2 The ReadUp Application

In addition to the Web page ReadUp applet, a search-oriented standalone ReadUp application has been developed. When started, this application displays the search panel shown in figure 9. The user may adjust the sensitivity of the search, and decide whether to display only the top-scoring document (the default), or all documents which exceed the search score threshold.

Figure 9: The ReadUp application's search window.

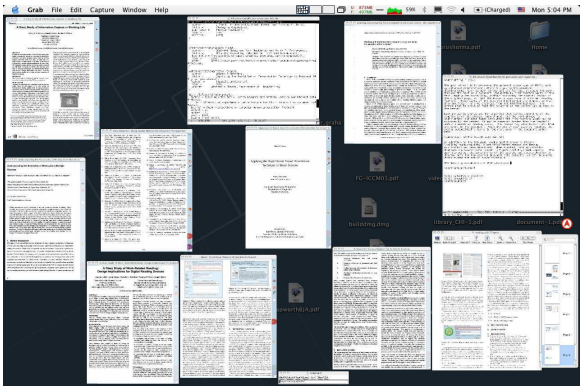

Figure 10: ReadUp windows on a user's work surface.

The matching documents are then displayed in individual top-level ReadUp windows. These windows are resizable, automatically scaling the ReadUp display to accommodate the reader's desired page size, and can be toggled between single-page and two-page display modes with a function key. More documents can be retrieved from the library by searching for them, using any of the existing open documents to initiate the search. Combined with a good window manager, this can provide a very useful way of working with a set of documents. Figure 10 shows a computer desktop during the preparation of this paper, using the MacOS X window manager's Exposé view. There are a number of open ReadUp windows containing various source documents for this paper, edit windows for the paper itself and the bibliography, and a PDF proofing window for the paper. The UpLib portal icon can be seen at the very far right, in the center.

5 Observations and Future Work

Two years of experience with UpLib revealed a number of interesting deficiencies in our initial design. Many (most?) documents users want to save seem not to be on their file systems, but rather available as Web pages or email attachments, or as paper. This however is in accord with our belief that any single user interface for document capture will be insufficient for all users, and that a large number of capture tools will always be necessary, to support the diverse collection of workflow microcultures present in every document-using society.

The drag-and-drop approach used in the UpLib Portal seems to provide a useful low-effort general-purpose interface for incidental personal document capture. We wonder if this interface could be adapted for larger group digital libraries, in university or corporate settings. In this context, documents dropped onto the portal would be forwarded to professional curation staff as suggested additions to the library. The search functionality of the Portal could be used as is in the larger group setting.

Interfaces for useful collection management are an active area of research. The latest version of UpLib's basic Web interface for collection management adds various date-related and textual display options. UpLib itself also provides new document icon generation algorithms that convey a better sense of the document's relative physical size, giving the reader an additional perceptual cue for document recognition.

The ReadUp document reader combined with an UpLib repository now provides readers with a standard annotation-supporting page-oriented consistent reading interface for documents of any format, which users seem to appreciate. Small touches such as page-turn animation, remembering which page the user was on the last time they opened the document, and effective text search, seem to provide major usability benefits. We have had reports of readers putting documents into UpLib not because they especially want to save them for the future, but because they want to read them with ReadUp instead of Adobe Reader, or their Web browser. The advantages of a ReadUp-style reading interface do not seem limited to personal digital libraries (though it clearly gains leverage from the projection-oriented design of UpLib); this approach could be easily used with larger group libraries.

We are currently exploring a range of more complex document management interfaces for the repository, and additional display modes for ReadUp. As improvements to document analysis and augmentation capabilities are made, we expect these interfaces to continue to change. When reading with ReadUp, all user actions, such as page-turning or searching, are optionally logged back to the user's UpLib repository. We are exploring ways in which this usage information can be exploited to give the user a more personalized, or easier to use, interface. In addition, we are looking at ways to confederate users' individual repositories, allowing a user to search over their colleague's collections.

6 Acknowledgements

We'd like to thank Eric Bier for many substantial conversations and feedback about the operation of the system, and Olga Gurevich and Lauri Karttunen for development of the linguistic technology used in the ADH display subsystem of ReadUp.

References

[1] Htmldoc, 2005. See http://www.htmldoc.org/.

[2] Apple Computer. Spotlight: Find anything on your Mac instantly., 2004. See http://images.apple.com/macosx/tiger/pdf/ Spotlight_Tech_Preview_20050111.pdf.

[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] B. A. T. Brown, A. J. Sellen, and K. P. O'Hara. A diary study of information capture in working life. In *CHI '00: Proceedings of the SIGCHI Conference Human Factors in Computing Systems*, pages 438–445, 2000.

[5] V. Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, 1945.

[6] 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.

[7] S. K. Card, G. G. Robertson, and W. York. The WebBook and the Web Forager: An information workspace for the World-Wide Web. In *CHI '96: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1996.

[8] 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.

[9] R. Dornfest. Google your desktop, October 2004. See http://www.oreillynet.com/pub/a/network/ 2004/10/14/google_desktop.html.

[10] P. Dourish, W. K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. B. Terry, and J. Thornton. Extending document management systems with user-specific active properties. *ACM Trans. Inf. Syst.*, 18(2):140–170, 2000.

[11] P. Dourish, W. K. Edwards, A. LaMarca, and M. Salisbury. Presto: an experimental architecture for fluid interactive document spaces. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 6(2):133–161, 1999.

[12] N. Ducheneaut and V. Bellotti. E-mail as habitat: an exploration of embedded personal information management. *interactions*, 8(5):30–38, 2001.

[13] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff I've Seen: a system for personal information retrieval and re-use. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 72–79. ACM Press, 2003.

[14] W. N. Francis and H. Kucera. *Brown corpus maunal: manual of information to accompany a standard corpus of present-day edited American English, for use with digital computers*. Brown University, Providence, Rhode Island, 1964.

[15] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. MyLifeBits: fulfilling the memex vision. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 235–238. ACM Press, 2002.

[16] J. J. Hull and P. E. Hart. Toward zero-effort personal document management. *Computer*, 34(3):30–35, 2001.

[17] D. Huynh, D. Karger, and D. Quan. Haystack: A platform for creating, organizing and visualizing information using RDF. In *Proceedings of the Semantic Web Workshop, The Eleventh World Wide Web Conference 2002*, 2002.

[18] W. C. Janssen. Collaborative extensions for the UpLib system. In *JCDL 2004: Proceedings of the Fourth ACM/IEEE Joint Conference on Digital Libraries*, pages 239–240, June 2004.

[19] 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.

[20] W. C. Janssen. Readup: A widget for reading. Technical Report TR-05-03, Palo Alto Research Center, April 2005. http://www.parc.com/janssen/pubs/TR-05-3.pdf.

[21] 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.

[22] C. C. Marshall and S. Bly. Saving and using encountered information: Implications for electronic periodicals. In *CHI '05: Proceedings of the Conference on Human Factors in Computing Systems*, April 2005.

[23] D. B. Noonburg. Xpdf. See http://www.foolabs.com/xpdf/.

[24] 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.

[25] 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.

[26] The Apache Project. Jakarta Lucene Overview, 2003. See http://jakarta.apache.org/lucene/docs/index.html.

[27] Xerox Corporation. *XDOC Data Format Technical Specification*, 1995.

[28] Xerox Corporation. Xerox FlowPort, 2004. See http://www.xerox.com/go/xrx/equipment/ product_details.jsp?prodID=FlowPort2.