Design and Implementation of the BoW User Interface

Dror G. Feitelson and Alex Kremer
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

November 14, 2001

Abstract

BoW (Bibliography on the Web) provides users with facilities for browsing and contributing to an on-line bibliographical repository. While the user interface is essentially based on well-known principles, their adaptation to the Web environment has presented some problems. The implementation experience has exposed some deficiencies of the HTTP protocol and of browser implementations that make such applications unnecessarily difficult.

1 Introduction

The Internet has evolved into the de-facto standard for the implementation of large-scale distributed applications, in which users around the world interact with servers to obtain information, make on-line purchases, and communicate with each other. This has led to the rapid design and implementation of new protocols with many new features. It is also accompanied by a very short development cycle, in which usability testing and learning from experience are often skipped in favor of rapid time to market.

The alternative is to settle for slightly older technology which has a higher degree of compatability with browsers and servers. In BoW, for example, we make heavy use of JavaScript, but at this stage do not yet use XML parsers. Rather than focus on leading-edge technology, we emphasize the adaptation of conventional principles of user interface design to the Web environment. For example, in order to reduce the latency of user actions the number of server interactions must be kept low; the challenge is then to use commonly available features of HTTP and JavaScript in order to do so.

The next section briefly introduces the BoW application. Section 3 explains the design principles that lie at the base of our work. Section 4 then reports on our experience in trying to implement these principles.

2 The BoW System

BoW (Bibliography on the Web) is a system for the collection of bibliographical information in a limited scientific domain [2]. It is based on the “field of dreams” approach: the system is essentially a collection of tools that provide an enabling technology, and leaves it up to the users to supply the actual content[1] [6]. As such, it is actually not limited to scientific literature, and can in principle be applied to any situation in which data items need to be classified and rated. A similar approach, relying on users to provide understanding, is used in the AntWorld search engine [4].

BoW is fashioned around a hierarchical topic index that spans the field being addressed. This is like the table of contents of a book: it partitions the field into major top-level topics, and within each one, continues the division into subtopics of ever finer scope. The overall structure is a DAG rather than a tree, as subtopics can belong to more than one branch. Apart from pointing to sub-topics, pages in the index can also contain direct links to bibliographical entries. Users who deposit new entries into the repository are prompted to link them to appropriate topic pages. This is supported by an automatic search that provides suggestions for appropriate topics [3]. An entry can be linked to any number of topic pages, as the user sees fit.

Apart from adding entries, users can also add links to existing entries that they encounter, add annotations to entries, and provide ranking (the details of how this is done so as not to burden the users are among the topics of this paper). The ranking is always within the context of a specific topic, as opposed to ranking according to citations [5]. Thus it provides information about *relevance* and not just about quality; in fact, the same document can have a high ranking in the context of one topic, and a low ranking in another. The ranking is used in the organization of topic pages, with entries listed according to their ranking, and ranks are also displayed in a concise form using happy or sad faces. Ranking makes the maintenance of the site a collaborative effort, in which each user benefits from the work and experience of previous users.

A possible usage scenario proceeds as follows. Initially the system is in normal mode, allowing browsing of the hierarchical topic index. Any entry that is displayed in this process can be ranked and marked for export. It is also possible to augment it with an annotation or links. Adding an annotation is done using a small form that appears in a separate window. Adding links is done by moving to link mode, in which browsing is used to find related topics and entries to link. Adding new entries is done with a special type-specific form, like adding an annotation; when the form is submitted, the system enters search mode, and displays related topics to which the new entry may be linked.

The main characteristics of this application are the following. First, it is interactive: work is accomplished by interacting with the system, requiring an extremely low response time. The goal is that system mechanics will not interrupt the normal flow of a user's work, and that it be responsive and natural to use. Second, it is collaborative: value is created by the actions of all users, and each benefits from the tracks left by others. This includes not only explicit contributions, but also implicit ones; for example, choosing to export an entry

[1] However, it is necessary to prime a new system with enough content to make it sufficiently attractive to new users.

adds to its rank even if no explicit ranking is performed. Finally, it is distributed: users are remote from each other and form the server, and may suffer from the limitations of HTTP and browsers.

A prototype system containing some 3000 bibliographical records from the field of parallel systems, organized into a topic index of about 140 pages, is accessible at http://www.bow.cs.huji.ac.il. The implementation is in PHP, with the data stored in a postgress relational database. But note that not all the features discussed here are already fully implemented.

3 Principles of the User Interface

Designing a user interface for a web application is similar to the design for a desktop application, and most of the same principles apply [8, 7]. However, the distance between the client and the server on the web requires special care to be taken. In the following we consider the BoW user interface design in the light of three important resources: screen space, communication with the server, and the user's time to learn how to use the system.

3.1 Utilizing Screen Real Estate

A well known maxim attributed to Ben Shneiderman is that "a pixel is a terrible thing to waste". Packing information and controls onto a limited screen area is perhaps the most crucial aspect of graphical interface design. The familiar desktop metaphor is very misleading in this respect: you can put way more on a physical desk than on a computer display.

In BoW, the basic screen layout is divided into five parts: three for data and context, and two for control (Figure 1). At the top is a general toolbar for system-level operations, such as searching, adding new entries, and exporting selected entries. Next come two frames that provide context. On the left is the topical hierarchy, which is shown in the familiar style of a directory tree. On the right is a listing of entries in the currently chosen topic. The two frames are linked to each other using color coding: the background color of the topic contents frame is also used to highlight this topic in the hierarchical display.

The bottom of the screen is devoted to the current entry. Again, color coding is used to link this with the listing of the topic contents. Most of the space is devoted to data about the entry itself, including its bibliographical details and whatever annotations and links have been contributed by users. At the left is a toolbar for operations that are specific to this entry, such as adding an annotation, displaying its data in BibTeX format, or making a correction. In addition, it is possible to mark the entry for export and rank it.

3.2 User and Server Interactions

Paraphrasing Shneiderman's maxim, which emphasizes the importance of seemingly minor things, we can say that a *click* is a terrible thing to waste too. One aspect of this is that it is desirable not to waste the user's time: if an effect can be achieved in one click, why force two? Another consideration, which is particularly important in a Web environment, is that clicks typically involve a lengthy server interaction. Such interactions should be reduced as much as possible, because they interrupt the interactive flow of the user's work.

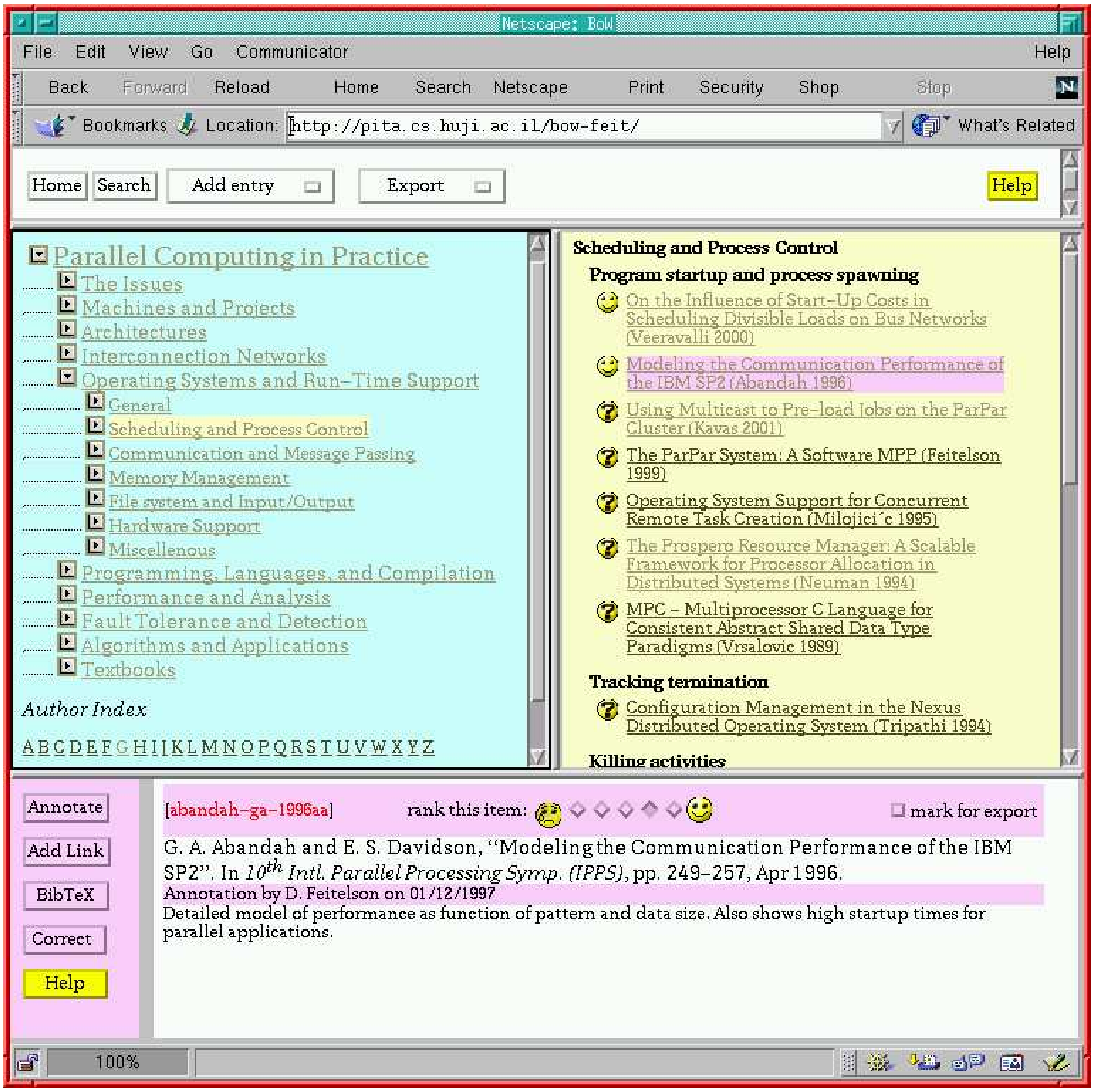

Figure 1: *The BoW interface in normal mode provides two levels of context, detailed data on the current entry, and both global and entry-specific controls.*

In BoW there are several instances where a click does more than one thing. Simple examples are correcting or annotating an entry. This is done in a separate window that contains a special form for this purpose. Submitting this form does three things: it sends the data to the server for storage, it closes the additional window, and it reloads the entry with its new data. Only in cases of error does the window stay in place, to allow for an error message and for a re-submit.

Server interactions may be avoided by clicks that only have a local effect. For example, in BoW ranking an entry or marking it for export are local operations that are done in the browser. Thus a user can use these features almost absent mindedly, without worrying about a disruption of his work. In other words, users should feel they are directly benefiting from everything they do, rather than having to take a detour in which they give up some of their time in favor of the system. Contrast this with the customary "add to cart" in practically all e-commerce sites, which forces the user to view the cart contents, or feedback and ranking facilities that force the user to fill out a form.

Obviously, however, the data must eventually be sent to the server. This is done by piggybacking it on another interaction. In our case, all new ranking data is sent to the server whenever a new topic is selected for display. This allows the server to take the new ranks into account in future displays. Export data is only sent to the server when the actual export operation is performed, together with the requested export format. The only case in which the data is sent to the server by itself is upon unload, when the user exits the BoW site.

3.3 Maintaining the Big Picture

While connectivity is at the heart of the Web, practically all Web information services treat their data as totally fragmented: each item for itself. In BoW, by contradistinction, we believe that context is all important. This is evident from the basic design and its emphasis on the hierarchical topic index. However, this level of context is not unique, and exists in most Web directories.

A more unusual feature is the way BoW treats keyword search. Most information services respond to a keyword query with a long list of matching data items, sorted by perceived relevance. Some systems, like ChaCha, group the data items according to their source [1]. BoW takes this an additional step: the keyword search produces a list of *topics* rather than entries. These are topics that contain many entries that match the query, and thus provide good starting points for browsing the concept(s) that actually underlie the query. Broad queries can be expected to lead to a high level, general topic, whereas focused queries should lead to narrower topics deep within the hierarchy.

While our search algorithm is fully developed and tested [3], it is not yet integrated with the prototype.

3.4 Reducing the Learning Curve

The need to learn how to use a new system is always a barrier to its adoption. In order to reduce this barrier, it is imperative that users be able to use the system for simple and common tasks with little if any tutoring. In BoW this is achieved using three means. When the system is first loaded, the screen space normally devoted for showing topic contents and entries is instead used for simple instructions regarding how to achieve such displays. This is all that is needed for browsing the system. During browsing, additional actions are possible using clearly marked controls (e.g. "mark for export"). Finally, specific help is provided where things cannot be guessed (Figure 2), and a help button providing access to the complete help system is always visible in the top toolbar.

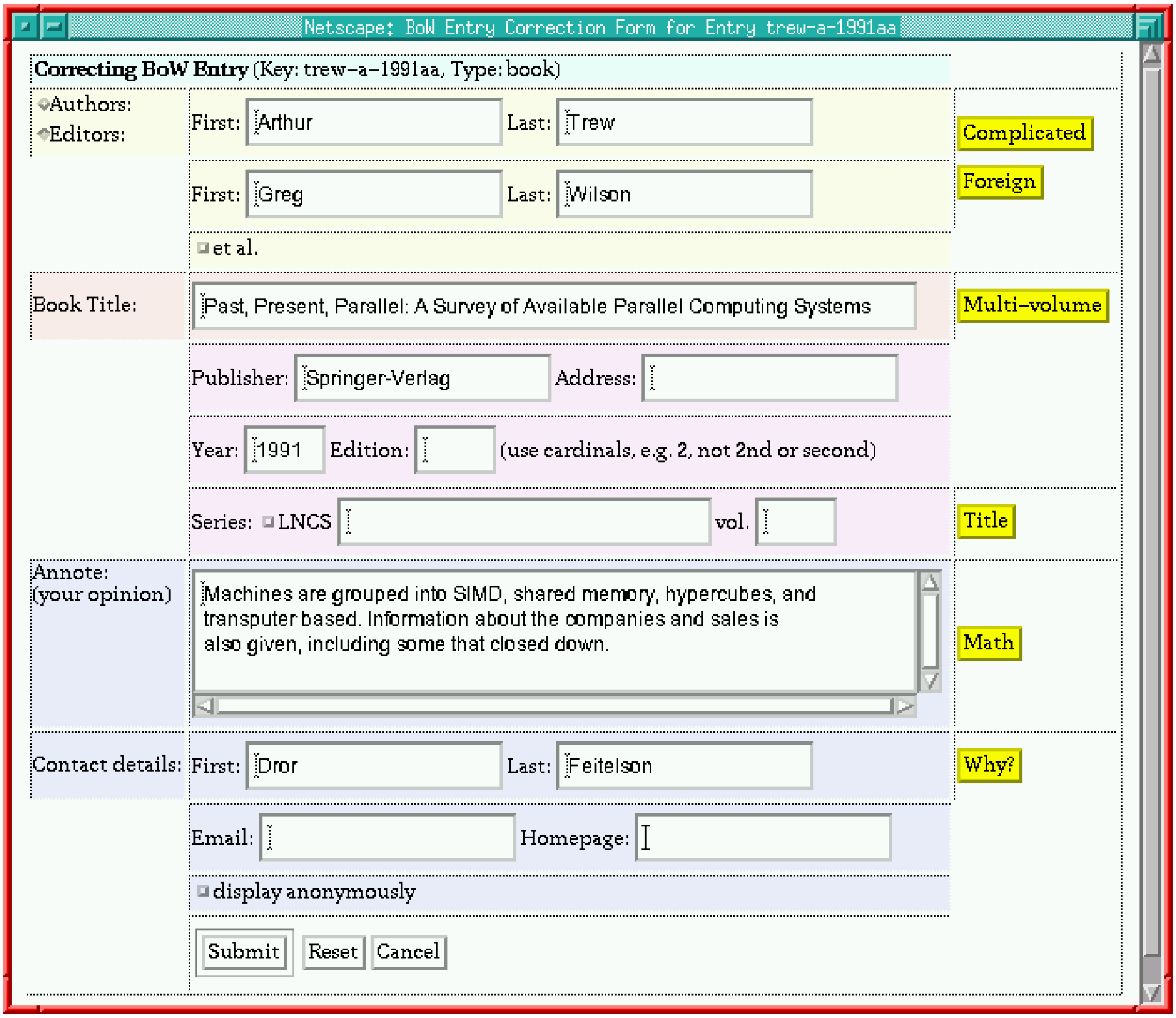

Figure 2: *The form for entering new data and correcting existing data contains specific help for different fields. Users only learn about these special cases when they need them.*

Another aspect of not wasting the user's time is building on experience (we can say that experience is a terrible thing to waste 8-). Human beings are creatures of habit, and it is extremely important to allow habits to be formed, preferably unconsciously. This then allows users to work without explicitly thinking about the technicalities of what they are doing, freeing their minds to the real work [7].

One aspect of this principle in BoW is that the basic screen layout of five frames is the same in all operating modes; only the contents of some of the frames, and possibly their relative sizes, may change. Thus in link mode, which allows links to be added to the current entry, the entry in question is copied to the top toolbar and displayed together with a set of controls for adding links (Figure 3). The current topic's contents are modified slightly, by adding checkbuttons next to headings to which links may be made. Finally, the bottom toolbar with operations on the current entry is replaced by an explanatory message. But the

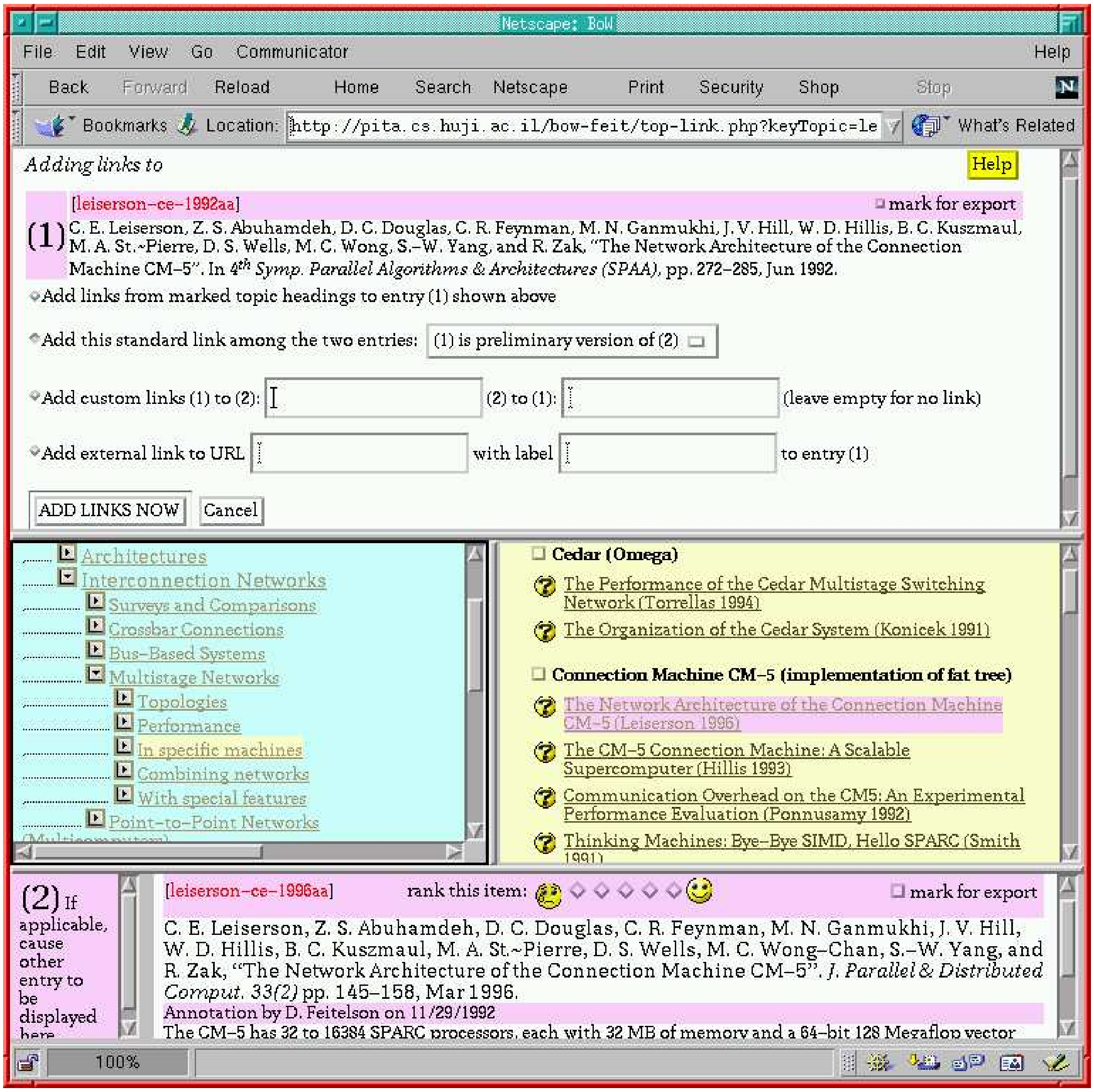

Figure 3: *The BoW interface in link mode uses the same basic layout as normal mode.*

layout and color coding stay the same. This is true for search mode and add mode as well.

In short, the key to forming habits is consistency. This is not limited to the overall design, but should also apply to details. An example is the consistent color coding of help using bright yellow. This is true for the general help buttons in the toolbars, the help entries in menus, and the context help next to fields in forms. Another example is that most operations are mode-independent. Thus entries can be ranked and marked for export whenever they are displayed, regardless of whether this is part of normal browsing or of searching.

Helping the user to learn how to use the system is important. But it is also important to identify things you cannot expect the user to learn. For example, BoW users are expected to link new entries to the hierarchical topic index, but cannot be expected to know this whole structure well enough to enable thorough linking. The solution is to provide automated help. Whenever a new entry is introduced to the system, the contents of the entry are used for a keyword search. The topics found are then presented to the user as suggestions for linking. The final decision stays with the user, based on human understanding of the actual content.

4 Implementation Experience

The implementation of many of the abovementioned features is straightforward. But the implementation experience also led to some observations regarding seemingly minor deficiencies of HTTP and browsers, and to some developments that may be useful for other applications.

4.1 Saving State

A common problem in the implementation of distributed Web-based applications is that of saving state information, in order to make the server aware of sessions despite the stateless HTTP protocol. This is commonly done using cookies and hidden data fields in forms.

BoW also has the opposite problem, of keeping information continuously in the browser. As mentioned above, ranking and export actions by the user are not propagated immediately to the server so as not to disrupt the flow of work. An additional benefit is that when an entry is re-displayed, its previous ranking can be shown, along with an indication of whether it has already been marked for export; this is simply done by checking the appropriate radio or check button, and also gives the user the opportunity to make changes. But in order to be really effective, this data should be provided for *all* entries ranked or marked by the user, not only the recent ones that have not been propagated to the server yet.

Our solution is to keep this information is JavaScript variables defined in the top level window, and accessed using set and get functions. In principle, this data has to be downloaded only once at the beginning of a session, and can then be used continuously as long as the user does not leave the site. However, due to difficulties in modifying the framesets when changing modes in a way that is compatible with all browsers, the current implementation re-loads this data whenever the operation mode is changed.

4.2 One-Way Communication

As noted above, we wish to avoid the disruption of normal work in order to send data regarding ranking or export to the server. This is achieved by collecting this data, and sending it to the server as part of another, unavoidable interaction. While this works, it has the undesirable side effect of cluttering the implementation: For example, the script whose main function is to load a new topic must also handle the storing of ranking information.

The root cause of the problem is the strict request-response cycle that lies at the base of HTTP. BoW thus serves as an example of an application that would benefit from the possibility of sending data to the server *without* waiting for or expecting a response. For example, the only solution we found for sending the data that has not been stored yet when the user leaves the site is to open a small temporary window, whose only function is to serve as the target for the server's response to the request to store the data. We don't need this response, but we cannot avoid it. In consideration for the user, we use the response to close this window. But even so, its brief appearance is annoying.

4.3 Separating Location from Loading

A more subtle issue is the HTML/JavaScript feature of loading a new Web page by assigning its URL to the location property of a window or frame. Our experience indicates that being able to separate the assignment from the loading would be beneficial.

One such benefit would be the possibility of creating better bookmarks. In multiframe sites like BoW, getting a certain configuration can require many intermediate steps. In each step, the contents of a single frame are replaced. The problem is that the original URL of the toplevel page does not reflect this, so creating a bookmark only enables one to return to the initial step, not to the final outcome. It is possible to pass arguments to this page, indicating what should be loaded into the sub frames; we indeed do so in BoW in order to restore a previous state when switching among modes. Using this mechanism at every step would then create a URL that can indeed be bookmarked with the desired results. However, such usage would have the very undesirable result of re-loading the whole page and all its frames each time that only one should actually be modified. This contradicts the principle that server interactions should be limited as much as possible to compensate for the distributed and relatively slow nature of the Web.

Making the location a writable variable that does not have the side effect of re-loading the page solves this problem: as each frame is loaded with a new page, all it has to do is to modify the location variable. When the user encounters a state that is worth saving, the browser's bookmark button can be used to copy the current contents of the location variable to the bookmark file. This also avoids the security concerns associated with allowing JavaScript direct access to the bookmarks file: all the JavaScript can do is assign the location variable, and only the user can initiate the action of copying this data to the bookmarks file.

A related problem occurs when, as is often the case in BoW, a form submitted from one window or frame eventually leads to a re-load of another. For example, the form for adding an annotation should eventually cause the frame displaying the entry to reload, showing the entry with its new annotation. The simple solution is to specify the entry frame as the target of the form, and to return the desired information (i.e. the full entry) from the form handling script. However, this causes the form handling script to become the URL of the entry frame; if the user subsequently uses the browser's reload button, the form handler will be invoked again instead of invoking the entry display script.

Again, the preferred solution would be that the form handler return the desired data directly, and that this include a snippet of JavaScript that would modify to location variable and point it at the right place. As this is currently impossible, we are forced to use a two-step approach. The form loads a small script that sets the location variable, and this has the side effect of loading the correct data. The price is a double server interaction where one would suffice.

4.4 Dynamic Web Pages

BoW makes rather extensive use of dynamic Web pages, even though it does not use any fancy graphics. One example is the "multi-action" clicks, where one click has several outcomes, e.g. selecting a topic in the hierarchical topic index both displays its subtopics in the same frame, and displays its entries in the topic contents frame. Another is the color coding that links frames: when the topic is selected in the index, it is also endowed with a background color that matches the background color of the topic contents frame.

The support for dynamic modifications to HTML pages is inferior by far to the available support for the implementation of desktop graphical user interfaces. The problems are well known. One is the lack of consistency in the handling of different objects on the page. Another is the lack of compatability between competing browsers (in fact, a large fraction of the reference literature used for development is devoted to tracking and circumventing such incompatibilities).

In BoW we decided to avoid using Java to add dynamic behavior, so as not to reduce the pool of potential users. This left us with JavaScript as the main programming tool, and the option to re-load pages as an escape route. For example, after experimenting with a downloaded JavaScript library that implements the selective display of a hierarchical index, we opted for a reload of the page from the server as a much simpler and safer solution.

It is ironic that in a rather ambitious system like BoW, it was simple features like the correct background coloring of selected text that caused the most grief. This could be done in isolation, but not when combined with a hierarchical structure that had to keep correct indentation when presented in a narrow frame, and also show a graphic icon at the beginning of each item (a face for the list of a topic's contents, or a button for the topic hierarchy itself). Finding ways to get these features to work correctly and robustly on different browsers has been a constant source of agitation, and in fact remains one of the more problematic aspects of the BoW implementation.

4.5 User Identification

Collaboration implies that many users contribute to the BoW site. One form of contribution is annotations. As anonymity breeds irresponsibility, we want to be able to identify the authors of such annotations. But users may also have legitimate reasons to preserve their privacy, and make comments anonymously. The solution adopted in BoW is to identify authorship to the system, but be able to request that it be withheld from other users.

A more radical design choice is that users will not need to log into the system, or provide passwords. When a user first loads a page from BoW, he is assigned a hard-to-guess random user ID, which is stored in a cookie. This ID thereafter identifies this user, at least as long as he does not change browsers or computers. Initially, this ID is not associated with any other information. Only when the user first contributes data is he asked to provide a name and contact information. These are then stored in the server's database together with the ID. They may be used by the site editor in case of subsequent disputes between users. Note that this design can be made more secure than the prototype implementation by using encryption when submitting the forms with user data. This would essentially prevent the danger of eavesdropping and masquerading.

5 Conclusions

Systems such as BoW attempt to use the Web infrastructure to implement interactive, collaborative, distributed applications. This stretches the limits of common Web technology. While the Web easily provides global connectivity, it does so using the rather rigid HTTP protocol, and within the confines of HTML facilities for display.

Our experience indicates that in many cases it is possible to employ conventional principles of user interface design, and adapt them for the Web environment. JavaScript largely suffices for saving state, achieving multiple application operations in response to a single user action, and reducing the communication with the server. But some minor modifications to HTTP and HTML will go a long way towards making it easier to implement such applications using the Web's infrastructure.

Acknowledgement

This work was supported in part by the Ministry of Science, Culture & Sport.

References

[1] M. Chen, M. Hearst, J. Hong, and J. Lin, "*Cha-Cha: a system for organizing intranet search results*". In 2nd *USENIX Symp. Internet Technologies and Systems*, Oct 1999.

[2] D. G. Feitelson, "*Cooperative indexing, classification, and evaluation in BoW*". In *7th IFCIS Intl. Conf. Cooperative Information Syst.*, O. Etzion and P. Scheuermann (eds.), pp. 66–77, Springer-Verlag, Sep 2000. Lect. Notes Comput. Sci. vol. 1901.

[3] M. Geffet and D. G. Feitelson, "*Hierarchical indexing and document matching in BoW*". In *Joint ACM/IEEE Conf. Digital Libraries*, Jun 2001.

[4] P. B. Kantor, E. Boros, B. Melamed, V. Meñkov, B. Shapira, and D. J. Neu, "*Capturing human intelligence in the net*". *Comm. ACM* **43(8)**, pp. 112–115, Aug 2000.

[5] S. Lawrence, C. L. Giles, and K. Bollacker, "*Digital libraries and autonomous citation indexing*". *Computer* **32(6)**, pp. 67–71, Jun 1999.

[6] R. W. Lucky, "*New communications services — what does society want?*". *Proc. IEEE* **85(10)**, pp. 1536–1543, Oct 1997.

[7] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems.* Addison-Wesley, 2000.

[8] B. Shneiderman, *Designing the User Interface.* Addison Wesley, 3rd ed., 1998.