Handling Backtracking in Web Applications

Bettina Biel, Matthias Book, Volker Gruhn, Dirk Peters, Clemens Schäfer
Chair of Applied Telematics/e-Business, Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany
{biel, book, gruhn, peters, schaefer}@ebus.informatik.uni-leipzig.de

Abstract

A common challenge in the development of web applications today lies in the handling of unforeseen navigation steps initiated by the user through the browser's Back, Forward and Reload buttons. These operations break the synchrony of dialog states on the server and the client, provoking non-intuitive and possibly destructive application behaviour if not properly handled. We therefore present an approach to handling Back and Forward navigation that realizes undo/redo semantics and illustrate its implementation using the example of a web-based conference management system. The presented approach is subsequently discussed with regard to its general applicability and alternative handling semantics.

1. Introduction

Web-based user interfaces (UIs) have become increasingly popular for client-server applications since they avoid a number of issues that window-based graphical UIs face in this sector: Web-based interfaces can be easily implemented in such a way that they can be displayed on a wide variety of hardware and operating system platforms, and they do not require any code to be executed on the client besides a web browser. The latter characteristic is probably the most important benefit, since it relieves users from the need to download and install any additional software on their system, which they may be reluctant or unable to do because of insufficient skills or security considerations. Web-based UIs are also attractive out of technical considerations, since they allow the construction of ideal thin clients that only render a user interface from a given description, but rely on the server to run the actual application and host all logic.

This is an important difference between client-server applications and window- or web-based UIs: While window-based applications still require some presentation logic on the client that reacts to events and handles them using callbacks to the server-side business logic, a web-based application runs completely on the server. The UI displayed on the client merely consists of pages rendered according to specifications (e.g. in Hypertext Markup Language (HTML)) generated by the server-side presentation logic. Due to the stateless nature of the Hypertext Transfer Protocol (HTTP) that is responsible for communicating data between the presentation logic and the browser rendering the user interface, their separation goes so deep that all user activities would seem like unrelated events to the presentation logic if developers did not implement additional measures to establish a session context spanning multiple pages and processing steps.

In the following subsections, we first show how the decoupling of UI generation and rendering can introduce dialog synchronization issues that lead to challenges when handling backtracking, and give an overview of related approaches to this problem. In section 2, we then present our approach to the identification of backtracking and reestablishment of dialog state synchrony, using the concrete example of the Paperdyne Conference Management System. In section 3, we will discuss generalizations and limitations of this approach, and present an overview of other ways to handle backtracking in web applications.

1.1. Dialog Synchronization Issues

The decoupling of UI generation and UI rendering would not be a problem if all possible ways of interacting with the UI could be specified at the time of page generation, i.e. if the links and buttons on a page would be the only widgets through which the user can interact with an appliction. On the web, however, this is not the case, since browsers often provide a number of additional widgets that enable users to interact with the application's user interface in uncontrollable ways. Typical features offered by browsers are:

1. Closing the browser window or requesting an external page[1]

1 We define an *external page* as any page not generated by the application's presentation logic.

2. Requesting a page through a previously set bookmark or a link from an external page
3. Cloning the browser window to obtain two user interfaces for the same session
4. Clicking the Reload button
5. Clicking the Back or Forward buttons

This way, the user can effectively perform the following operations that the presentation logic can neither allow nor forbid:

1. Leaving the application
2. Jumping to a certain page out of context
3. Performing parallel operations in different contexts
4. Repeating the request for the current page
5. Recalling previously rendered pages

These operations may cause serious problems for the application—not so much because they are unpredictable and uncontrollable, but rather because they break the synchrony between the server- and client-side state of the dialog system.[2] Usually, the client-side dialog state is determined by the server-side dialog state through the presentation logic's generation of pages to be rendered. Thus, the presentation logic always knows which state transitions it can currently expect to come in from the client. The above operations, however, change the client-side dialog state in a way that either cannot be detected or is not expected by the presentation logic (with undetected changes leading to unexpected changes sooner or later) [7].

Since these operations are not uncommon (clicking the Back button, for example, is the second-most frequent user activity after clicking on a link [3]), they cannot be disregarded as special cases, but should be treated as properly as regular clicks on links. They should at least be handled gracefully, i.e. in a way that does not break the system, but leads it to a well-defined error or fall-back state. While not really satisfactory from the user's perspective (since the page he intended to reach is not displayed), this mechanism is implemented in most applications today. However, it would be even better if backtracking could not only be handled gracefully, but actually *sensibly* in the current application context (where the definition of "sensible" depends on the reaction that the user would intuitively expect from the system; usually displaying the previous page). While a graceful handling should be feasible on a purely technical level, a sensible handling poses more of a challenge since it must take the application semantics into account.

2 We define the *dialog state* as the currently displayed page and the set of possible transitions to other dialog pages.

Of the uncontrollable operations listed above, only the first one is uncritical: Even when the application logic cannot detect right away that a user left the application, a timeout mechanism can sooner or later trigger all necessary steps to terminate his session. A graceful handling of the second case (jumping to a page out of context) can be accomplished by redirecting the user to a well-defined starting point, such as the application's home page. Alternatively, for a semantics-aware handling, the system needs to decide if the requested page can serve as a valid entry point into the application, and possibly allow the direct jump.

Parallel operations in cloned windows (case 3) pose both a technical and a semantic challenge: Firstly, even if the presentation logic is capable of distinguishing user activities in the different window instances (i.e. if it can manage multiple client-side dialog states in the same session), it still needs to deal with potential semantic conflicts (e.g. when the user is assuming different roles with different privileges in different windows). We are still examining the implications of this scenario and are working on dialog control algorithms for managing it.

1.2. Challenges when Handling Backtracking

In this paper, we will focus on the handling of cases 4 and 5—repeating recent requests for the same or previous pages through use of the Back, Forward and Reload buttons. While these operations break the synchrony between client- and server-side dialog state, they do so according to the known semantics of stacks, which enables the presentation logic to use the same semantics for solving the two-fold problem of identifying their occurence and re-synchronizing the dialog states.

However, the problem is exacerbated by the fact that backtracking may occur in two ways, one of which is undetectable by the server: When the user clicks the Back button, the browser may either resend the previous request to the server, where it can be handled by the presentation logic; or the browser may simply retrieve the previously displayed page from its local cache without contacting the server. The decision whether to resend the request or just redisplay the cached page depends on a number of factors such as the HTTP method (GET or POST) used to send the request, any caching directives contained in the page's header, and the user's configuration of the browser's cache size and caching strategy [4]. There does not seem to be a way that eliminates caching altogether and reduces the backtracking problem to the simpler case of resent requests. With local caching, however, the user may backtrack a number of steps before the server becomes aware of it, so the unsynchronized dialog states may diverge by a number of steps before the server has the opportunity to fix the problem.

Note that even when the browser resends requests during

backtracking (so that the presentation logic becomes aware of it right away), handling the repeated requests is still not trivial, since they may not simply lead to a page that displays information, but might also trigger the re-execution of certain application logic operations that could cause side effects by changing the data model. In order to avoid potentially destructive re-executions of certain operations (e.g. duplicating transactions), the presentation logic may need to intercept resent requests, skip all application logic operations and only display the page that ultimately ends the dialog step.

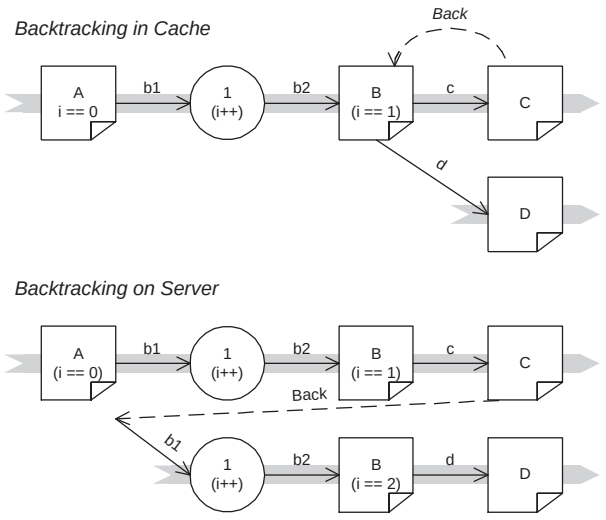

Figure 1. Backtracking in Cache vs. on Server.

Figure 1 shows both the case of backtracking in the cache and on the server in the Dialog Flow Notation (DFN) [2], using solid arrows for requests and dashed lines for clicks on the Back button. To illustrate the breaking of the dialog sequence caused by backtracking, the sequence continues on a new timeline in those instances. If we assume that the contents of a variable i are displayed on pages A and B and incremented in operation 1 in between, backtracking in the cache from C will render B with the same value of i as before. However, if backtracking from C repeats the previous request, the server-side operation incrementing i is repeated, leading to a page B with a different value for i.

1.3. Related Work

Back navigation on the web is discussed in many publications, however mainly describing user behavior when browsing and searching the web, and proposing new implementations of browsers' Back and history functionality.

Shubin and Perkins [8] analysed conflicts within the conceptual model developed by users working with the desktop instead of the web, and their resulting impression of backtracking. As Tauscher and Greenberg have found [9], 30 percent of web navigation activities are Back button clicks. This reflects the results of Catledge and Pitkow [3], underlining the importance of the problem.

From the browser's point of view, Cockburn and Jones [4] analysed navigation facilities provided by web browsers, and described the resulting usability problems. They propose browser extensions such as dynamically adapting to the users' browsing actions. Cockburn and Greenberg [5] propose several recency-based behaviors as alternate implementations for backtracking.

Regarding backtracking from the server's point of view, Baresi et al. [1] use assertions to specify the navigation problems resulting from the use of the Back button. At amazon.com, for example, a user who adds a product to a shopping cart and then goes back might accidently add another product. They present a solution to this problem using assertions that specify web operations: By defining low-level (atomic) pre- and post-conditions and high-level (non-atomic) property descriptions, designers should decide whether a parameter of an operation is set by the current web page, the current internal state of the application, or whether both states have to be identical to start an operation. After illustrating their idea using an OCL example for the Back button problem, four implementations are proposed. They range from the inspection of the content of a page to determine if it has become stale, via the prohibition of browser stacks, to exact definition of pre- and post-conditions for an operation, and finally to a decision that the actually shown information is always correct and therefore controls the application.

Motivated by an explicit user-centered perspective, our approach corresponds with the latter idea. It differs in that we have not used assertions, as Baresi et al. did, but mirror the stack on the server. This way, our approach is able to return to the old state and its stored data.

2. Handling Backtracking in Paperdyne

In this section, we present how web navigation with Back and Forward buttons is enabled in a wizard-style dialog used to send e-mails with customized form letters by the conference management system *Paperdyne* [6].

2.1. Sending E-Mails in Paperdyne

Paperdyne is a web-based system supporting the organization of the technical program of scientific conferences. Paperdyne is used by authors to submit scientific papers, by referees to download these papers and to write reviews, and

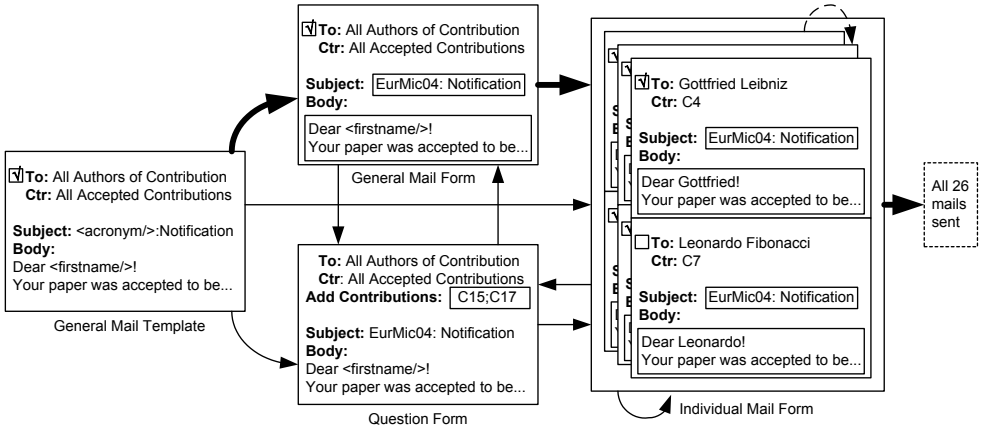

Figure 2. Paperdyne Mail Process.

by the program committee chair (PCC) to organize the submission and reviewing process.

A major task of the PCC is to communicate with authors and referees in different situations and distribute information. In order to assist the PCC with these communication tasks, Paperdyne offers a variety of e-mail templates for standard situations. For example, it is the job of the PCC to notify authors of accepted papers and include the review comments in this e-mail. Although this is a standard e-mail, the PCC might want to add conference-specific details about preparing a camera-ready copy, as well as individual comments for authors he knows personally.

This process of sending standard e-mails is basically simple: A *general mail template* is transformed into a conference-specific *general mail form*, where global tags (such as the conference acronym), are replaced (see Figure 2). In this general mail form, the PCC can make global changes. In a second step, each individual mail can be edited in the *individual mail form*. Here, individual tags such as the authors' names are already replaced. Within the individual mail form, the number of e-mails might be too large to be all displayed on one page. An additional navigation mechanism is therefore necessary to browse between the e-mail pages.

The process becomes more complex if the PCC decides to add or remove recipients in any step. Paperdyne allows to add users (or referred papers) in the *question form* (see Figure 2). Thus, the number of steps in the process of sending e-mails is not limited. Since users are likely to click the Back button sometime while completing it, the system must provide special mechanisms to handle it.

In the architecture that was designed to solve this problem, we have two major components. The front-end *GUI and request-handling component* generates the web interface and handles and evaluates requests, while the back-end *process component* performs all necessary mail transformations. We will describe both components in detail in the following sections.

2.2. The Process Component

The process component's purpose is to perform transformations of e-mail contents during the process of sending e-mails. This component relies on an XML document that includes mail content and process information.

Initially, the process component receives a general e-mail template (see Figure 2) and performs a first transformation to one of the other three template types, depending on the process information. All other transformations are initiated by the front-end. The type of transformation is based on the transition, defined by a user interaction and the process definition of the XML document. In the transformation, conference- and user-specific data is inserted into the XML document, with general tags being replaced by content as early as possible (in Figure 2, for example, the conference acronym can be inserted within the first transformation, but user names cannot be fetched before the user is known). Thus, not only the process transformation is implemented in this back-end component, but also the process state and accessory template data management.

When clicks on the Back or Forward buttons are detected by the front-end, the undo semantics of the Back button affect the process state of the process component, so the back-end is obviously not stateless anymore. Our solution for handling backtracking in web applications is the introduction of *back and forward stacks*. Handling these stacks must be a task of the front-end, because only the front-end can identify and synchronize stack depths with the stack in the user's browser, but since back-end states are affected, the stacks must be stored within the back-end. In our design, the complete state is held on the stacks in XML documents that contain process information. With these sim-

ple stacks, undo and redo functionality can be implemented within the process component.

To realize the undo semantics, the intuitive behavior of the e-mail component is to move the current state to the forward stack and and replace it with the top-most state of the back stack (the redo behavior works vice versa). Unfortunately, client-side caching functionality can prevent the application from recognizing right away that the Back or Forward button was clicked. This makes it difficult to distinguish between reloading and submitting a request. For the backend, the difference lies in the fact that the forward cache needs to be deleted when the Submit button is clicked.

On the other hand, it is not really necessary to figure out which of the buttons was pressed: As long as the user does not change data in the web form, it is not dangerous to assume that the Forward button was clicked, even if the event was caused by clicking the Submit button. All the back-end component needs to do is process the requested transformation, compare the result with the forward stack and delete the whole forward stack if the transformation result is different from the top of the forward stack. Of course, with this approach, the back-end may still keep the forward stack although the forward cache is not available in the user's browser anymore. This is no problem, however, since the forward stack on the server will be deleted upon the user's next submission of a page.

The back-end only provides the trivial functionality of the two stacks that are used by the front-end, which is responsible for managing them according to the back and forward stack semantics. After a brief discussion of the interface between the back-end and front-end, we will explain the front-end's use of the two stacks in the following subsections.

2.3. The Interface between the Process and GUI Component

In the last subsection, we showed how transformations within the e-mail process can be modeled in XML documents. The front-end needs some of the information in the XML documents, too. An easy way of providing this information would be to let the front-end component access the XML document. However, this solution would have some serious disadvantages.

While the XML document for the back-end contains process information and mail content, the front-end does not need the process information. Of course, this problem could still be resolved by just passing a sub-document to the front-end. However, doing this would cause unnecessary dependencies between front-end and back-end: The Document Type Definition (DTD) is designed to allow easy transformations within the process. Changes within this DTD must

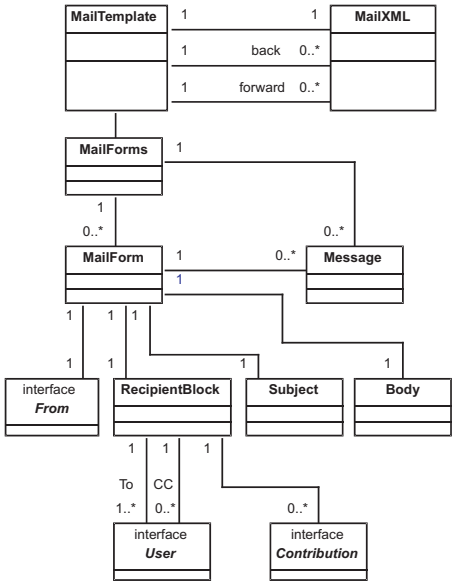

Figure 3. E-Mail Class Diagram.

be possible without having to adjust the front-end component.

In Paperdyne, the front-end interface is therefore designed as a tree of objects (see Figure 3), not as an XML document. The advantages of XML are not needed here since it is an internal interface. On the other hand, we need features like event handling. When passing Java objects, it is possible to send change events to the back-end and let it perform syntactical and semantical checks. Especially within the general mail form, where XML tags are used as placeholders, these checks are useful.

The main back-end class is called `MailTemplate`, and the root class for the interface object tree is called `MailForms` (see Figure 3). The necessary child objects, like `Subject` or `Body`, are associated with the `MailForms`-Object with the same structure, as this information is displayed in the front-end. The back- and forward stacks are managed within the `MailTemplate` class as XML documents. This class is able to generate the `MailForms` tree with the content of `MailXML` documents. Changed data is inserted into the `MailForms` tree by the front-end, and the back-end is able to merge this information into XML documents, which are held by the `MailXML` objects. With this structure, we are able to generate the necessary change events and semantical checks.

2.4. The GUI and Request-Handling Component

We previously discussed the implementation of the back-end part of Paperdyne's e-mail functionality. Now, we describe the front-end part in more detail. We depict how the

information provided by the back-end is used to render the web pages and how the user's interaction with these pages is handled.

To generate a web page, we assume that we are given an instance of the `MailTemplate` class, which represents the content of one page. The associated `MailForms` object allows to iterate over all instances of the `MailForm` class and to traverse the tree-like structure shown in Figure 3. The different instances of the `MailForm` class represent separate sections on the final web page and thus determine the overall layout of this page. Every `MailForm` object comprises a static `From` field, which indicates the sender of the actual mail. Additionally, a `Subject` and a `Body` object have to be rendered as single or multi-line text input controls. The `RecipientBlock` consists of one or multiple entities which refer to users (cf. `User`) or contributions (cf. `Contribution`). These entities are either check boxes or text input controls for receiving textual queries. During the rendering process, all input controls are uniquely named in order to make them identifiable for later operations. Additionally, the content of the `Message` objects is added to the web page as static content.

Depending on the state given by the `MailTemplate` object, the user is allowed to invoke different actions. Hence, the appropriate Submit buttons are added to the web page.

So far, we showed how the data structure given by the `MailTemplate` object is rendered as a web page. This web page is presented to the user in the web browser, where he can enter data using the form controls. When submitting the form by pressing one of the Submit buttons on the page, an HTTP request is sent to the application. Now, the corresponding instance of the `MailTemplate` class has to be identified, which is needed to process the request. Therefore, every generated web page is given a hidden input field at rendering time. This field contains a unique number, which can now be used to search for the corresponding instance of the `MailTemplate` on the forward stack and the back stack. After the correct instance has been found, the stacks are re-arranged in such a way that the found instance of the `MailTemplate` object becomes the top element of the forward stack.

Once the corresponding `MailTemplate` object has been identified, the same traversal through the object structure as for rendering the web page can be applied: The naming scheme used for the input controls at rendering time can now be used to extract information from the HTTP request and to pass the extracted information to the elements of the `MailTemplate` object by calling the respective setter methods.

Finally, the type of action the user has invoked by pressing one of the Submit buttons can be determined by parsing the request information, and the corresponding method of the `MailTemplate` object can be executed. The outcome of this operation (a new `MailTemplate` object) can be compared to the current instance of the `MailTemplate` on top of the forward stack. In case of equality, the request results from a reload of the web page. In this case, the user has pressed the browser's Reload button. If the two objects are not equal, the user has invoked a different operation. Then the forward stack is cleared and the current mail template is put on this stack. The new web page can now be rendered again as described above.

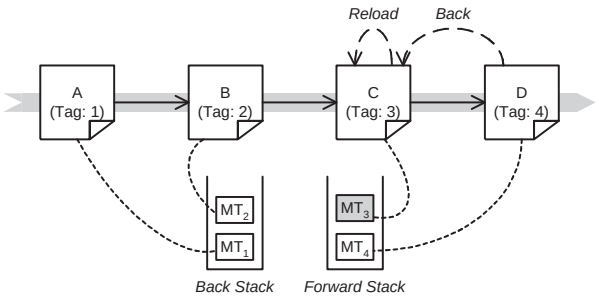

Figure 4. Back and Reload Example.

To clarify this behavior, we consider Figure 4. Here, the user has executed an example dialog, visiting the masks A to D. The forward and back stacks have been populated accordingly with the mail templates MT_1 to MT_4: the templates MT_1 to MT_3 are lying on the back stack with MT_3 on top, and MT_4 is on the forward stack. Now, the user makes use of the Back button of his browser. Hence, the mask C is displayed to him (typically retrieved from the browser's cache). Next, the user reloads this page. The system recognizes this situation and re-arranges the stacks in a way that the mail template MT_3 becomes top on the forward stack as shown in Figure 4.

Continuing from this situation, we can imagine a situation where a user who has navigated to mask C and reloaded this page, performs another back operation leading to mask B. Then the user performs an action which leads him to mask E. This behavior is depicted in Figure 5. As mentioned before, by performing the action which leads to mask E, the mail template instances MT_3 and MT_4 are removed from the forward stack. This behavior is correct since the transition from mask B to mask E as accompanied by a change of the data basis which makes backtracking to the pages C and D impossible.

3. Discussion

After discussing how backtracking is handled in Paperdyne, we now abstract from this concrete showcase in order to generalize the approach presented in this paper.

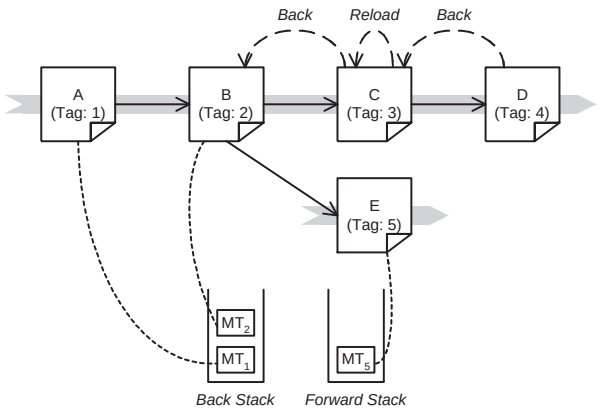

Figure 5. Navigation Example.

3.1. Generalization

Generalizing the type of data structures (in our case, `MailTemplate` instances) on the stacks is trivial. The only requirement for this is that all generated pages must be tagged with unique identifiers and that these tags can be associated with instances of the data entities on the stacks.

In our showcase, the transformation of the data structure into a web page implies a correspondence between the object structure and the structure of the rendered page. However, we can imagine other scenarios where this correspondence is not given and the dialog data is structured in a different way than the web page. Nevertheless, the creation of a web page should also be no problem in this case, even if some more effort is required.

The stack mechanism used in Paperdyne is a crucial element for establishing the backtracking functionality. Therefore, a similar mechanism must also be provided in any general solution. One problem that could make this approach infeasible is memory consumption, hence scalability issues must be considered. We have to be aware of the fact that for every user logged into the system, we need at least one stack per session for the handling of backtracking issues. On these stacks, several instances of dialog objects have to be stored. Fortunately, we can assume the number of these objects to be limited, since usually the dialog flows which require handling of backtracking will comprise a rather small number of steps. Therefore, only few objects have to be stored on the stacks for every user. Usually, these objects can also be rather small, so that this approach—although it is memory intensive—should be feasible in a more general setting as well.

In Paperdyne, there is typically only one user (the PCC) who is provided with the backtracking facility. In this aspect, Paperdyne stays behind the requirements for the general solution, since the number of users needing the backtracking functionality is very limited. But on the other hand, the objects which are stored on the stack in Paperdyne are very large and memory-consuming (in certain cases, there can be hundreds of mails, contained in plain text in the structure). In typical applications (such as web shops etc.), these objects will probably be significantly smaller compared to Paperdyne, so that we are confident that the scalability of our solution can be ensured for other types of applications as well. Nevertheless, further evaluation is needed to prove this conclusion.

3.2. Backtracking vs. Undoing

In the approach presented in this paper, the application data is stored on the stack together with the description of the pages, and therefore changed consistently with the user's backtracking (as illustrated in the top half of Figure 6). This way, we actually implemented an undo/redo mechanism: By returning to a previous page, the user also returns to a previous version of the entered data, from where he may either initiate a new operation or go forward again using the browser's Forward button (which restores the data of the following step). This behaviour makes working with the application very intuitive, since the user can rely on the fact that the data displayed on the client is the same that is stored on the server even when he is switching back and forth between pages.

Note, however, that not all operations are undoable: In the Paperdyne example, we are employing a multi-step dialog "wizard" to collect all necessary data (message template, recipient group, individual recipients, individual messages etc.) for the execution of a certain transaction (sending a form letter by e-mail). While the preparatory data-collection steps can be undone (in the top part of Figure 6, for example, removal of the user b is undone by the first backtracking event, and a is removed by the user instead) the transmission of the e-mails obviously can not. The presentation logic must be aware of this "point of no return" and disallow any backtracking beyond it, for example by displaying an error page or leading the user to the wizard's initial page, from where he can start a new transaction.

In other applications, however, these undo semantics may be undesired and actually counterintuitive. For example, in an online shop, users may navigate between various overview and detail pages to browse the shop's inventory. After putting an item a into the cart, which leads them to the shopping cart page, they may return to the shop using the Back button. It would be counterintuitive to undo the "add to cart" operation at this point—rather, the updated cart contents should now be visible on all pages, even when the user is backtracking (as illustrated in the bottom part of Figure 6). Despite the different semantics for backtracking, we also have a "point of no return" in this scenario: After the user purchases the items in his cart, going back to pre-

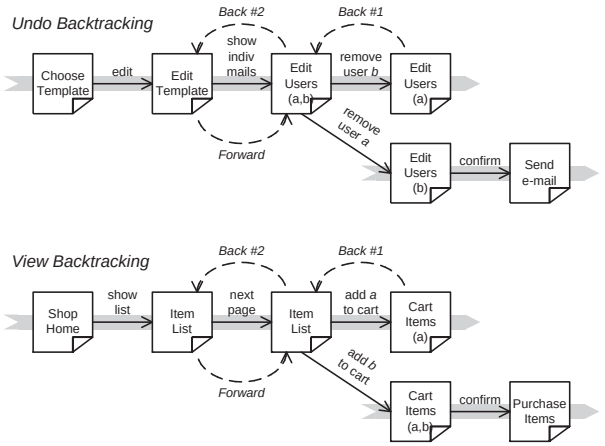

Figure 6. Undo vs. View Backtracking.

vious pages can neither undo the purchase nor display the contents of the shopping cart—instead, the user can only begin a new transaction with a clear cart.

Technically, the difference between this "undo backtracking" and "view backtracking" is determined by the mechanism used to store and retrieve the displayed data: For undo-backtracking, it needs to be stored on back and forward stacks with associated page identifiers, and retrieved from this stack according to a mechanism that mirrors the browser's backtracking logic. For view-backtracking, the data needs to be stored in a way that allows updates to overwrite the previous values, so the presentation logic can always retrieve the most current value. The mechanism for detecting backtracking and identifying the page that the user reverted to remains the same for both approaches, since the presentation logic needs to resynchronize its dialog state with the client even when it is not undoing operations.

4. Conclusion

We identified a number of challenges induced by the fact that web applications cannot only be navigated by the means provided by themselves, but also by navigational aids provided by web browsers—namely, the Back, Forward and Reload buttons. To fix the breach of synchrony between client and server dialog states they may cause, we presented an approach that holds past dialog states in so-called back and forward stacks on the server, so the server can revert to previous states according to navigation on the client. Our discussion of this approach showed that it can be used in any application that aims to implement undo/redo semantics for the Back and Forward buttons.

5. Acknowledgments

The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

References

[1] L. Baresi, G. Denaro, L. Mainetti, and P. Paolini. Assertions to better specify the amazon bug. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 585–592. ACM Press, 2002.

[2] M. Book and V. Gruhn. A dialog flow notation for web-based applications. In M. Hamza, editor, *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*, pages 100–105. The International Association of Science and Technology for Development (IASTED), ACTA Press, 2003.

[3] L. D. Catledge and J. E. Pitkow. Characterizing browsing strategies in the World-Wide Web. *Computer Networks and ISDN Systems*, 27(6):1065–1073, 1995.

[4] A. Cockburn and S. Jones. Which way now? analysing and easing inadequacies in www navigation. *International Journal of Human Computer Studies*, 45(1):105–129, 1996.

[5] S. Greenberg and A. Cockburn. Getting back to back: Alternate behaviors for a web browsers back button. In *Proceedings of the 5th Annual Human Factors and the Web Conference*, 1999.

[6] L. Johnston, D. Peters, J.-G. Schneider, and U. Wellen. Requirements analysis in distributed software engineering education: An experience report. In *6th Australian Workshop on Requirements Engineering (AWRE 2001)*, 2001.

[7] H. Shubin and M. M. Meehan. Navigation in web applications. *interactions*, 4(6):13–17, 1997.

[8] H. Shubin and R. Perkins. Web navigation: resolving conflicts between the desktop and the web. In *CHI 98 conference summary on Human factors in computing systems*, page 209. ACM Press, 1998.

[9] L. Tauscher and S. Greenberg. Revisitation patterns in world wide web navigation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 399–406. ACM Press, 1997.