Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 200 (2008) 119–131

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Controlling Control Flow in Web Applications

Robin Message[a,1] Alan Mycroft[a,2]

[a] *Computer Laboratory*
University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK

Abstract

Control flow is often key problem in current web applications. For example, using the back button gives a POSTDATA error, using multiple windows books the wrong hotel, and sending a link to a friend does not work.
Previous solutions used continuations as a model for user interaction. However continuations are insufficient as a model of all web interactions. We believe the protocol and browsers themselves are insufficiently powerful to represent the control flow desired in a web application.
Our solution is to extend the protocol and browser sufficiently that these problems can be avoided. We seek to be agnostic about how web applications are written and instead recognise that many of the problems stem from underlying weaknesses in the protocol.
As an example, the application ought to be able to inform the browser that pressing back on a payment confirmation page is not allowed. Instead, the cached page can be displayed in a read-only, archive fashion to the user, or a new page can be shown instead which is consistent with the global state.
We discuss how some of these ideas may be implemented within the existing HTTP/1.1 protocol; and what modest extensions to the protocol would enable full implementation. We also discuss the interaction with Web 2.0 and the security and privacy implications of our extensions.

Keywords: continuations, control flow, navigation, protocol, state, web application, browser, AJAX

1 Introduction

The web was originally designed as a read-only, hypertext medium for publishing structured documents [5]. Since then, it has evolved, seemingly at random, to become one of the predominant user interface systems currently in use. Web servers have become more powerful and quickly moved from providing read-only access to files to now providing write access to databases, stock markets and shipping companies. Similarly, web client software has not stood still and has added such things as opening multiple windows, tabbed browsing, cookies and bookmarking. An increase in collaboration between web users has also led to a desire to share links that will continue to work for the foreseeable future.

[1] Email:Robin.Message@cl.cam.ac.uk

[2] Email:Alan.Mycroft@cl.cam.ac.uk

doi:10.1016/j.entcs.2008.04.096

Many of these things exhibit bad feature interactions with each other. Feature interactions were first described by Dworak et al. in relation to telephone systems [7]. Since then, feature interactions have been found in most complex software systems. In our case, the features of the web browser and the web application are the source of the interaction. To give some examples:

(i) The web application uses cookies to create the impression of a stateful protocol. However, the user has opened multiple browser windows, and so generates responses that the application misinterprets. For example, opening two windows containing products in an online store and clicking buy on the first one to be opened may result in the second one being bought. Added to this, are two browser windows the same as two tabs in the same window? [3]

(ii) Taking the URL of a page that you have to be logged in to see and giving it to a friend will sometimes not work, even if the friend has access to that site with their own login. Indeed, bookmarked pages often do not work even for the same person, even if they log back in.

(iii) Clicking refresh or forwards or backwards results in an error message being displayed to the user about POSTDATA in many web applications. Since the back and forward buttons were designed for navigating stateless pages, it did not matter that they resent the same request. However, since POST requests that can change the state of the server have been added, it has become necessary to alert the user that they ought not go back and forth in web applications, since this will cause these POST requests to be resent, and therefore the server state will be changed multiple times. However, there is no reason in principle why a user should not go back and forth in the application except that the protocol is poorly designed.

In order to work around these problems, various ‘solutions’ have been attempted. On the one hand, many sites tell users not to use the back/forward buttons, not to open multiple windows and not to make bookmarks. They then implement rather draconian measures (for example, ending a user’s session and forcing them to login again and start what they were doing from scratch if they use a prohibited feature) as a forcing device to ensure that users do not use these browser features. While this system works, it would be preferable to enable the user to use browser features whenever it can be made safe for them to do so.

Queinnec suggests that web site state should be held for the user in continuations and resumed when a user navigates to a new web page [10]. This simplifies web programming control flow and is the approach adopted by the more permissive web application designers, and it is a good model for some web applications. However, it is flawed as a general model for all web applications for the following reasons:

(i) The implementation of continuations in commercial languages like Java requires either a preprocessor and some kind of suspend/resume control flow, which is ugly, or the use of a thread per user session, which can be suspended

[3] Throughout this paper we refer simply to multiple browser windows, since they seem to behave the same way as tabs in all mainstream browsers and we avoid the messy “window/tab” construction.

and resumed to simulate the action of a continuation. However, this is not scalable since commercial languages use operating system level threads and operating systems place comparatively low limits on the number of threads.

(ii) Even if continuation objects are supported natively, storing them is also a problem. If they are stored on the server, storage required will grow indefinitely, or continuations will have to be timed out, which will be annoying for the user. If the client is required to store them, then this will be a huge leap backwards for readable URLs, since the URL will end up containing some non-human understandable encoding of the continuation. It will also increase traffic from the browser to the web server, since opaque continuation objects are likely to be far larger than existing URL-based descriptions of pages, and it damages one of the useful things about web applications – namely that you have access to all of your data anywhere you have access to a browser.

(iii) Since continuations only cover the variables local to each part of the application, they offer no help to the programmer on how to manage access to the global store. For example, once a purchase has been made online, returning to the "Are you sure?" confirmation page continuation surely cannot reverse the credit card transaction.

(iv) Continuations are not suitable to store all the data about a user's session since users can and do branch sessions and then merge the branches together again. We believe a framework should be agnostic about what to do in this situation, but continuations alone give no way to do this. For example, in a typical shopping application, a user opening multiple browser windows and clicking "add to basket" in each of them should result in one basket that can be taken to the checkout, not multiple baskets that each have one item in.

In our approach, we seek to embrace the script-centred or page-centred approach to web application design [6, p.4–6] and not to imposed a session-centred design on the programmer. Instead, we seek to improve the HTTP protocol so that the web application may better control what actions can be taken by the user's browser. Some improvements we suggest are:

(i) Allowing the server to express what happens when the user presses forwards or back.

(ii) Letting the server identify if the user is using multiple browser windows, and which windows are making which requests.

(iii) Changing the protocol so the POSTDATA error can no longer occur.

We look practically at how these extensions may be implemented, both within the current protocol, and also as extensions to existing browsers. Finally, we examine how these enhancements interact with Web 2.0, and what security and privacy implications they have.

2 Critical analysis of the HTTP protocol

In this section, we compare the HTTP protocol, and the web applications built on top of it, to well-understood existing protocols and patterns.

We define GET as an HTTP request that sends a URL to the server and receives an HTML page in response. A GET request can send user data from a web form, but this data is encoded in the URL, and the convention is this data must represent an idempotent request, for example a search term can acceptably be sent in GET request, but a request to delete an item cannot be. POST is an HTTP request that sends a URL and some data to the server and receives an HTML page in response. A URL is a Uniform Resource Locator which is the address of the resource being requested. Each request also sends a number of headers, which are key-value pairs and constitute meta-data about the request. A header containing a per-session nonce (called a cookie) is used for session handling in most web applications.

2.1 Web applications from the client-server perspective

Web applications can be seen as "client-server" applications. In a client-server application, one process runs the server, which provides a service, for example persisting data, and another process runs the client, which uses the service, for example by querying and displaying the server's data. In general, a server may accept connections from multiple clients, but a client connects to only one server at a time. For example, in the X Windows system, each window is a client and the X server itself is responsible for drawing each window onto the actual screen.

A user may use one or many clients at the same time to connect to the same server; however, when multiple clients are used, they tend to use the server in orthogonal ways. In the X windows example, a user may be using multiple clients, but each area of the screen belongs to only one client at a time, so the clients do not interfere.

Web applications can be seen from this perspective, however, there are a number of conceptual and implementation differences. We naturally consider the web application to be the server. If we consider the client to be the web browser, then the clients are no longer orthogonal, since a user can and will open multiple browser windows within the same application. This is the main conceptual difference between web applications and classical client-server applications.

In terms of implementation, web applications do not keep a connection open like most client-server protocols do, so each communication is inherently stateless. State is therefore built on top of the HTTP protocol, but this can lead to problems, for example, the possibility of application layer spoofing of sessions instead of having to spoof at the transport layer. In the case where one user has multiple windows open and each window is sharing the same session identifier (which is standard in all current browsers), the server cannot tell which window has sent a request. The server must therefore make no assumptions about the response that led to this request – in the case of bookmarking, that response could have been sent a long time ago. This lack of knowledge about windows has caused problems in the past, for example, a hotel website where opening multiple windows and clicking "book" causes the most recently opened hotel to be booked, not the one you clicked on.

The stateless nature of the protocol and the lack of client orthogonality lead us to conclude the client-server model is not a good fit for web applications.

2.2 Web applications in the Model-View-Controller pattern

Many web applications are developed using the *Model-View-Controller* (MVC) pattern [11], which is designed to help decouple user interfaces from the logic of the application. In the MVC pattern, the application is divided into a model, which holds all the data of the application, and one or more views, which provide user interfaces to the model. The controller manages creation and destruction of the model and the views, and coordinates input from the views into the model. The controller is also responsible for serialising access to the model by the views. Use of this design pattern in web applications, for example the Apache Struts framework [4], usually considers most of the code running on the server as the model and controller, and the view as the HTML pages sent to the client.

One problem with this use of the MVC pattern is that the controller is often marginalised and does not really exist at all. Without a controller, the MVC pattern essentially degenerates to the client-server model described above. However, the MVC pattern is a good way to think about how web applications should be written. The model is a good match to the server and the data on the server, and the concept of the view matches well to the browser windows, each describing a portion of the model. Where then is the controller? We believe the controller should be seen as part of and implemented on the browser, coordinating multiple views of the same application. This is a good abstraction, but is not perfect because it does not allow the controller to moderate access by different users to the same part of the model. Generally, web applications use optimistic concurrency control or no concurrency control at all, so it is not really a problem that the controller is per-user instead of per-model. It is certainly better than the current solution, which has one controller per view, since integrating the controller with the view is the same as having no controller at all.

When viewed from this perspective, several deficiencies in the HTTP protocol are obvious:

(i) Views in the MVC pattern observe the model, and update themselves when the model changes. This is typically done by the model notifying the views when it changes. This is not possible with HTTP, so the user must manually refresh the views when they know something has changed, or the views must poll the model.

(ii) Since there is no controller, nothing mediates the transactions between the model and views. For example, the model does not know when all the views of it have been closed.

Overall, the MVC pattern gives a good view of how web applications should be modelled.

2.3 Divergent state

HTTP is a stateless protocol. However, both the client and server have state in a typical web application. The server has some sort of global store and the client has local state consisting of a history of visited pages, cookies and any form fields that have been modified by the user but not submitted to the server yet. Since these multiple states exist, care must be taken to ensure that the state on the client and the server does not diverge—that is become irreconcilable. Examples of actions that cause divergent state would be going backwards in the browser to a page before an action is committed on the server, or editing the same record in two different windows and attempting to submit both windows. This second example highlights another problem—that the browser has not one state but a set of states, which may share some elements, corresponding to each window open.

2.4 POSTDATA errors

A POSTDATA error occurs when the user refreshes or otherwise renavigates to a page that was requested using an HTTP POST request. An error message is displayed to the user, informing them that they must POST the data again else the page cannot be seen. However, if the data is sent again, any action that was done in the response to the request will also be done again.

Another problem caused by POST is that any page that was created as a result of a POST cannot be bookmarked properly. For example, copying the address of a payment confirmation page and sending it to another computer, from which you want to print the confirmation page, will not work.

The essential cause of POSTDATA errors is the fact that the POST command contains within it an implicit GET command, but there is no way for the browser to redo that GET without doing the POST as well. It would be better therefore if the POST command could either fail and return the user to the form they were trying to submit, with an error message and all their data still intact, or it could succeed by giving the browser a new URL to GET.

3 An enhanced model

In this section we discuss a new model for web applications and how this model can be implemented on top of the existing protocol and browsers.

Our model is based on the ideas of client-server applications and the Model-View-Controller pattern. It involves the following seven extensions, most of which are based on adding extra metadata to the existing protocol:

(i) Each browser window is assigned a unique window identifier by the browser. These identifiers would be transmitted as a header in every request from the browser. A new browser window would additionally transmit the identifier of the window which it was opened from (i.e. its parent). The window identifiers should not be reused within one session of a web application, so a relatively short identifier would be sufficient.

(ii) The web application on the server is informed when a window is closed. This will enable additional housekeeping by the server to occur on demand, for example logging a user out automatically when they shut the last window that has a session with an application, and decrementing any reference counts held by that windows, for example in critical regions.

(iii) The server will inform the browser of the bookmarkability and sharability of a page. When the page is bookmarked, or the URL of the page or a link copied, metadata from the server can be used to control what happens. We envisage programmers marking each page as having one or more of the following statuses:

Not bookmarkable This item cannot be bookmarked and the user will be informed of this if they try. A textual description to the user could be given separately.

Bookmarkable This item can be bookmarked as normal.

This instead This item cannot or should not be bookmarked, but the server suggests some other page that can be bookmarked and is equivalent. Multiple pages could be suggested, each with a textual description, for the user to chose between.

Not shareable The user is informed that this link cannot be shared publicly. Optionally, the application may provide a link to the user which will take them to a part of the application where they can acquire a link that is sharable. Such a link may have to be paid for, or may have a time limit, extra advertising, or other restrictions.

Shareable This item can be shared as normal.

Share this instead When a link is being copied, the user is informed that if they want to share the link with other users, they should use some other link instead.

Alternative links for sharing can be seen in access control terms as capabilities – they carry both the authority to view an item and an identifier of the item itself. These options give a safe way of controlling which pages can be bookmarked and give the application flexibility to create behaviour that suits what the user expects.

(iv) Pages and links can be marked as *unclonable* and *this window only* respectively. These indicate to the browser that a section of the application is not to be open in more than one window at a time.

(v) In responding to a POST request (a request that may have side effects on the server), the web application may optionally request that the user's browser history is modified. Specifically, after taking some irreversible action, the web application will inform the browser which pages may have depended on state that has been modified by this action. If the browser has any of those pages cached, it may only display them to the user in a read-only fashion. The application may alternatively opt to replace pages in the history with new pages, for example changing a checkout screen to a list of what was in the user's basket before the checkout.

(vi) After a POST request, instead of sending a new page as a response, the server should inform the browser of a GET request it should make. This request will show the results of the POST, and since this is a GET request, it can be bookmarked, navigated to, and refreshed without causing the original POST action to occur multiple times.

(vii) Some way for the server to inform a browser window that the information it contains has been updated would be very useful but not essential to our proposals.

3.1 Implementation over existing browsers

Addressing each issue from the preceding section in turn, we consider how they could be implemented in existing browsers within the existing protocol.

(i) Giving a unique identifier to each browser window is partially possible within the existing protocol. Usefully, one of the headers sent by the browser is the referrer header which identifiers which URL the browser window was viewing before this one. Therefore, new windows can be identified by having a blank referrer, or a referrer from another site. Then, to continue tracking windows, each page is sent to the client with an extra identifier added to every link URL on the page. When a page is requested within an application, the extra identifier is checked to see if it has been used before. If it has not, then this can be treated as the new "main" window. It may actually be anew window, in which case the existing window will now be seen as a new window if the user navigates within it, but this is not so important to the server. If the server has seen this identifier before, then this could be a new window. However, if the user has used the back button to return to a cached copy of the original page, then this system will not work. One workaround to this is to make all the pages in a web application uncacheable. Another problem with this system is some users opt not to send the referrer header for privacy reasons. In general, this system could work in some cases in identifying multiple open browser windows, but is not ideal.

(ii) The server can be notified when a window is closed by javascript running within that window, but this may not work in all cases since javascript may be disabled. Additionally, since the HTTP protocol does not keep network connections open, it is impossible to ensure the server is notified of a crashed client, since there may be no communication channels open between the server and client at the time of the crash.

(iii) Javascript could be used to intercept some attempts at bookmarking or copying links, but in general control over bookmarking and copying of links will require direct support from the browser.

(iv) Marking pages and links as *unclonable* and *this window only* cannot be done without the browser interpreting extra meta-data. One thing that can be done (and is used commercially, for example by Amazon [2]) is to make most links a form submission instead. Since most browsers do not give the option to submit a form in a new window, this gives some of the needed functionality.

(v) Control of the user's browser history is currently unavailable to web applications for security reasons. This cannot be implemented on the client side. However, by marking all pages as uncacheable, the client will request the pages again when they try to navigate to them in their history. At that point, the server can detect that these pages are no longer valid and send a different page to the browser.

(vi) Removing the implicit GET request from inside the POST request is possible using the current protocol. By sending a "302 temporary redirect" response [8, pp. 62,134] to the browser, the browser will GET the new page specified, and will store that GET in its history instead of the original POST request.

(vii) Sending data from the server to the client is outside the scope of the current HTTP standard. The HTTP/1.1 standard does permit the server to send messages in a "chunked" encoding [8, pp. 23]. The chunked encoding allows the connection between server and browser to stay open, and the server to send multiple responses, each wrapped in is own "chunk". This would allow the server to update the browser, but browsers currently buffer "chunked" messages so they cannot be used in real-time. If browsers did not buffer the individual chunks, then it would be possible to use this to update windows as required. Another work-around[4] is to send an XMLHTTP request to the server with a long time-out. When the server has something to say, it replies to the request. If the request times out, it is resent. This is a good solution, but would be neater if there was no timeout and no need to restart the whole request in order for the server to send the next update.

Overall, all of these features can be implemented over the existing HTTP protocol, but some of them would require additional support from browsers.

3.2 Minimal browser enhancements required

In order to meet the requirements in Section 3, some enhancements to the browser are required.

Each browser window should allocate itself a unique id and transmit that as a header to the server when making requests. For privacy reasons, it would be best if the window allocated itself a new unique id when it moved to a different web application. When a new window is opened within the same application, the browser should send the identifier of the window which opened it as well as its own.

When a window is closed, it should send a request to the server informing it that is has closed. The server would have to provide metadata, either in the HTTP response headers or as part of the HTML, explaining what request should be made when the window was closed. This is slightly wasteful of bandwidth, so a standardised URL to open, or an alternate HTTP method, for example called CLOSE, could be used to remove this.

[4] We are grateful for this suggestion from a reviewer.

There is already a standard for specifying supplemental links from a page [1, sec. 12.3]. A standard for naming alternate bookmark and sharing links would need to be created and then browsers could use these elements to support the functionality described.

In order to allow the server to modify the page history, it could return a list of regular expressions in an extra header. Each expression is marked as either entirely removed, viewable read-only, to be removed from the cache or to be replaced with a different address. Any pages in the history that match the regular expression will be modified as requested.

A method of receiving chunked HTTP responses in a javascript callback function would enable dynamic web applications to update their views based on changes to the model, in keeping with the MVC pattern.

4 Extension to Web 2.0

Web 2.0 is a term used to mean many different things. A full explanation of all its meanings in given by O'Reilly [9]. In this paper however, we will restrict Web 2.0 to mean web applications that use Asynchronous Javascript and XML (AJAX) features. These enable code running within a web page to make additional requests to the server and update the view given to the user based on this additional data and/or computation done on the client. A Web 2.0 application may be based on a single web page but a more common paradigm seems to be to use AJAX for informing the user of additional data from the server that was not included with the original page, and for making reversible changes that are then committed. HTTP POST is still used to make irreversible changes. For example, AJAX may be used to suggest possible values that could go in a search field on a form, but the search is not done until the form is submitted and a new page is shown.

We now consider how each of our desiderata from Section 3 may affect Web 2.0 applications. In this section, when we refer to web applications, we shall mean ones making use of AJAX.

(i) Identifiers for windows are mostly irrelevant since most current applications use only a single window. However, being able to manage the complexity of a user having multiple views open would probably be aided by the server knowing about the windows that are open and what they are displaying.

(ii) Being informed of window closing would be very useful, since it would allow any pending actions to be reverted or saved so the user can continue from where they left off when they return to the application. This is especially important in a Web 2.0 application since there is likely to be a larger amount of uncommitted user data than in an ordinary web application.

(iii) Currently, a Web 2.0 application cannot easily be bookmarked, and links cannot easily be sent to others, for example, a links to a particular part of a collaboratively edited document. However, the proposal of sending meta-data with each page about how it can be bookmarked and shared would not apply to Web 2.0 applications, so a different method is needed. We envisage that the meta-data about bookmarking and sharing that is normally sent in the headers could also be changed by code running on the client as part of the Web 2.0 application. The application could also register some kind of event handler, so when a user right-clicks on something and selects "Bookmark this page", the context of where they clicked and the current state of the application could be used by a script to produce a more detailed bookmark.

(iv) Control over whether links open in a new window or if they are restricted to a single window would be equally useful in Web 2.0 applications as ordinary ones.

(v) Modifying page history would, like control over bookmarking, be very helpful in Web 2.0 applications. Currently, actions taken within a single page of a Web 2.0 application have no impact on the user's navigation history. The application should be able to programmatically access the history and add relevant items (similar to an undo history in an editor program). These items could be URLs to request from the server or scripts to run that would return the page to a particular state, perhaps even subject to constraints. For example, pressing back after the user has done some irreversible action could display a message explaining this to the user, without reloading the page.

(vi) Separating POST from GET does not really affect Web 2.0 applications except to make all parts of the application easily bookmarkable and refreshable, as in ordinary web applications.

(vii) The mechanism for communication from server to client is actually intended for advanced Web 2.0 application, for example an e-mail client, server monitor or collaborative editor. While several suggestions for implementation have already been made, a further option is to send a *multipart/x-mixed-replace* MIME type in response to an AJAX request. Such a response can then be used by the server to transmit extra information to the client at any time.

Of course, many Web 2.0 applications have gone the same way as older ordinary web applications and tried to restrict users from browser features they consider 'dangerous'. One project, Prism [3] aims to create a launcher for each web application that is almost indistinguishable from a native GUI application. In contrast, we feel that the browser idioms and features are useful and users are accustomed to having them, so it would be better to accommodate and improve them where possible.

We also feel there is more to be done in separating the Model, View and Controller in Web 2.0 applications, and while Web 2.0 has made it possible to view new data without downloading the View again (i.e. to update the Model), it would also be good if the View could be updated without that necessitating the Model also being reloaded.

5 Further Discussion

5.1 Making the HTTP protocol stateful

One further enhancement that would make sense would be create a stateful variant of HTTP. Currently, the HTTP protocol can keep a connection open after a request/response has occurred in order to run another request/response immediately afterwards for performance reasons. These are typically timed out quickly. However, this could be extended further and all communications within one web application could be run within one HTTP connection, that is kept open between requests, even if there is a long delay. This would also be useful in load balancing for very heavily used web applications. Currently, load balancing is done by sending the user to a random server each request they make, so session data must be stored centrally in order to give a coherent view to the user. By allocating a browser a single server at the start of the session, state can be kept on that server only[5], which would simplify the application at the expense of keeping more TCP connections open. It would also automatically show when all the windows of a web application are closed, or the network connection fails.

5.2 Security and privacy implications

The extra features we define in Section 3 are not designed to increase the security of web applications. In fact, their use could lead to careless programmers making web applications less secure. For example, programmers may assume that certain actions can not happen, when in fact they have simply asked the browser for them not to occur, something which could easily be circumvented. We hope to address this in the future by having a framework by which servers can enforce constraints, like *unclonable*, that are advised to the browser.

Additionally, these features have an impact on user privacy. However, this impact can be minimised in two ways. Firstly, each distinct web application (defined as the domain part of the URL) could be treated distinctly by the browser. For example, a different set of window identifiers could be used for each domain. This would prevent a user being tracked across multiple applications, even if the applications themselves are collaborating to try and see, for example, which users are using both applications. Only elements of a page fetched from the domain of the current web application would be given access to these extra features, which would restrict abuse. Secondly, in order to use these extensions, a server would send metadata saying this page was part of a web application with a certain base URL (which would need to contain at least the full domain of the page). The browser could then ask users whether they are willing to treat this site as a web application.

[5] State could also be replicated to a backup server in case this server goes down, but this could be done in the background after each request is served.

6 Conclusion

The motivation for this paper was to better understand the conceptual design and implementation of the HTTP protocol and its current usage in browsers. We compared the protocol to other similar protocols and examined existing approaches for creating web applications, identifying commonly encountered problems. We then proposed a series of extensions to the protocol, which make it far more suited to web applications. Several of these extensions require additional features of browsers; the next step is to implement these features and evaluate how they improve web applications.

Acknowledgement

This work was supported by an EPSRC studentship. Thanks to Alastair Beresford for helpful discussions and to the anonymous reviewers for their helpful comments, particularly for suggesting that we discuss Web 2.0.

References

[1] *HTML 4.01 Specification*, Technical Report REC-html401-1999-12-24, W3C (1999).
URL http://www.w3.org/TR/html401/

[2] *Amazon* (2007).
URL http://amazon.com

[3] *Prism* (2007).
URL http://labs.mozilla.com/2007/10/prism

[4] *Struts* (2007).
URL http://struts.apache.org/

[5] Berners-Lee, T., R. Fielding and H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, The Internet Society RFC 1945 (1996).
URL http://tools.ietf.org/html/rfc1945

[6] Christensen, A. S., A. Møller and M. I. Schwartzbach, *Extending Java for high-level web service construction*, ACM Transactions on Programming Languages and Systems **25** (2003), pp. 814–875.
URL http://doi.acm.org/10.1145/945885.945890

[7] Dworak, F. S., T. F. Bowen, C. H. Chow, G. E. Herman, N. Griffeth and Y. J. Lin, *Feature interaction problem in telecommunication systems*, in: *Proceedings of the Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, Bournemouth, United Kingdom, 1989, pp. 59–62.

[8] et al., R. F., *Hypertext transfer protocol – http/1.1*, The Internet Society RFC 2616 (1999).
URL http://www.ietf.org/rfc/rfc2616.txt

[9] O'Reilly, T., *What is web 2.0*, Technical report, O'Reilly Media (2005).
URL http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[10] Queinnec, C., *The influence of browsers on evaluators or, continuations to program web servers*, in: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (2000), pp. 23–33.
URL http://doi.acm.org/10.1145/351240.351243

[11] Reenskaug, T., *THING-MODEL-VIEW-EDITOR*, Technical Report 5, Xerox PARC (1979).
URL http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html