Delft University of Technology
Software Engineering Research Group
Technical Report Series

FireDetective: Understanding Ajax Client/Server Interactions

Nick Matthijssen, Andy Zaidman

Report TUD-SERG-2011-002

TUD-SERG-2011-002

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Paper accepted for the tool demonstrations track of the 33rd International Conference on Software Engineering (ICSE 2011).

FireDetective: Understanding Ajax Client/Server Interactions

Nick Matthijssen
Delft University of Technology
The Netherlands
nick8maal@gmail.com

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

Ajax-enabled web applications are a new breed of highly interactive, highly dynamic web applications. Although Ajax allows developers to create rich web applications, Ajax applications can be difficult to comprehend and thus to maintain. FireDetective aims to facilitate the understanding of Ajax applications. It uses dynamic analysis at both the client (browser) and server side and subsequently connects both traces for further analysis.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*reverse engineering*

General Terms

Human Factors

Keywords

Ajax, program comprehension, web applications, dynamic analysis

1. INTRODUCTION

Over the last decade web development has evolved from creating static web sites to creating rich, highly interactive web applications. The most important technology in realizing this shift is Ajax (Asynchronous Javascript and XML), an umbrella term for existing techniques such as JavaScript, DOM manipulation and the XMLHttpRequest object [3]. Since its inception in 2005, Ajax has gained in popularity and is now part of many websites, e.g., Google's GMail. Unfortunately, Ajax also makes developing for the web more complex. Classical web applications are based on a multi-page interface model, in which interactions are based on a page-sequence paradigm [6]. Ajax changes this by allowing asynchronous requests to be made after a page has been loaded and allowing JavaScript code to update parts of the page in the browser, i.e., making delta-updates without reloading the complete page.

Before the dawn of Ajax, Hassan and Holt already noted that "Maintaining web applications is problematic" [4]. The extra complexity that Ajax adds will certainly not improve this situation. Strangely enough, research efforts focusing on program understanding specifically for Ajax applications are scarce (e.g., [2]).

These observations, together with the rapidly growing number of Ajax web applications, motivated us to create FireDetective. Previously, we witnessed how web developers use a bottom-up approach to understand Ajax applications, which is often inefficient for understanding Ajax applications [5]. In this context, we created FireDetective, a dynamic analysis tool that traces both the client (browser) and server side and connects both traces for further analysis. A preliminary user study with 8 participants pointed out that FireDetective allows web developers to understand Ajax applications more effectively, more efficiently and with more confidence [5].

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05.

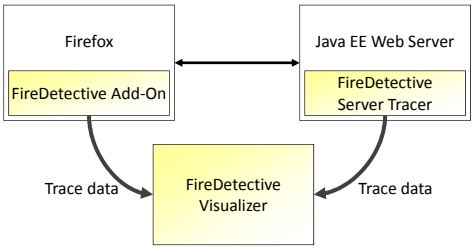

Figure 1: Architecture of FireDetective.

2. FIREDETECTIVE

FireDetective is a tool that records execution traces of the JavaScript code that is executed in the browser *and* of the server side code on the server. Subsequently, it "connects" the client and server side traces, effectively enabling the Ajax developer to see all information in a single trace, which, from the interviews that we held with Ajax developers is regarded as highly useful to get a good understanding of the control flow through an Ajax application [5].

The high-level architecture of FireDetective is shown in Figure 1. The tool consists of a Firefox add-on which records JavaScript traces and information about Ajax abstractions, and a server tracer which can be hooked into a Java EE[1] web server. Both of these components forward the data that they record (via sockets) to the visualizer, the third and final component of FireDetective. The visualizer then processes and visualizes the data in real-time. A benefit of this architecture is that it allows users to use Firefox to interact with an Ajax application, as they normally would, and then use the FireDetective visualizer to inspect what is going on "under the hood". Currently, the tool is built for Ajax applications with a Java + JSP back-end.

2.1 Linking Traces

FireDetective records information about abstractions that are specific to the Ajax/web-domain, such as (Ajax) requests, DOM events, timeouts, etc. This is a key element of the tool: it enables us to link the aforementioned execution traces in meaningful ways. These abstractions can also be used as familiar starting points for program

[1]Java Enterprise Edition. See `http://java.sun.com/javaee/`.

understanding, so-called *beacons* [8]. The Ajax/web domain abstractions that we use are [5]:

- Full page requests
- Non-Ajax requests
- Top-level script load invocations
- Web template invocations
- DOM events
- Ajax requests
- Timeouts

Some links between traces/calls and abstractions represent a causal relationship, e.g., some JavaScript call *causes* an Ajax request, which then *causes* a server side and – when the response is received – JavaScript trace to be created. By following these links in one direction, tool users are able to answer "what?" and "how?" questions about the program, e.g. "how was this DOM event handled?". Moreover, links can also be followed in the reverse direction, enabling tool users to answer "why?" questions, e.g. "why did this Ajax request occur?".

2.2 Interactive Visualization

The visualizer displays the collection of traces and abstractions to the user. Its interface is shown in Figure 2. The visualization's design is loosely based on guidelines outlined by Shneiderman [7]: information visualization tools should allow for creating overviews, zooming, filtering, and providing details on demand. This design correlates with a top-down comprehension strategy [8].

Three main views are used, each of which shows a different level of detail. The first view is a high-level view (annotated with "1" in Figure 2), which shows a tree representation of the aforementioned abstractions (except template invocations). Expandable tree nodes may reveal more detail, e.g., expanding an Ajax request node shows its relation to particular traces and calls, i.e. the life cycle of the request. The second view is a trace view which displays one execution trace at a time, as a call tree. Each tree node represents a single call, with expandable subcalls. The third view is a standard source code view.

The three views are linked: selecting a high-level entity in the first view shows the related trace in the trace view, and selecting a call in the trace view shows the related code. There is also one side view (annotation 4), which contains a tree representation of the resources (e.g., code files) of the Ajax application. Clicking a resource shows the file in the code view. The view can be filtered to only show the files that were used for the current page, which greatly reduces the number of files that are shown, and allows a tool user to quickly see which resources are involved on the current page. The user can also select a block of code (e.g., a JavaScript function) to highlight and cycle through invocations of that block of code in the high-level view and trace view.

A disadvantage of execution traces is that they can quickly grow to massive proportions [9]. In order to reduce the size of traces, we use two simple, well-known trace reduction mechanisms [1]. The first one is to filter out all library calls and only keep calls that are specific to the Ajax application that is being analyzed. Both client side libraries (such as Dojo[2]) and server side libraries (such as Java EE server internals) are filtered out. The second mechanism concerns stopping and starting recording. This allows the user to time slice the Ajax application, and, for example, to find out how a particular interaction with the Ajax application is handled.

3. IMPLEMENTATION & CHALLENGES

The various components of FireDetective are implemented in different languages, using different APIs. First, the FireDetective add-on is implemented in JavaScript, using the add-on API that Firefox provides[3]. We chose Firefox because it is well-known, and it provides a relatively mature platform for building browser add-ons. The add-on consists of about 2.2Kloc. The FireDetective server tracer is implemented in C++ (and a tiny bit of Java), a choice which was dictated by the tool interface that we use for tracing the execution of Java code. The server tracer consists of about 1.4Kloc. Finally, the visualizer is implemented in C#; it is the largest component in terms of lines of code: about 8.1Kloc.

3.1 Implementation Details

JavaScript function calls and Java calls are recorded using Firefox' debugger interface and the Java VM tool interface, respectively. This has the advantage that no code needs to be instrumented, and that the approach also works for JavaScript code that is generated dynamically and "eval"-ed on the fly.

The connection between browser and server is made by appending a custom header `X-REQUEST-ID` containing an id, to every outgoing HTTP request. Upon receiving the request on the server side, the id can be detected by the server tracer. DOM events are registered in Firefox by adding event listeners for all possible DOM events for the `window` and `document` objects. Ajax requests and JavaScript timeouts (and intervals) are registered by wrapping all related properties, functions (e.g., `XMLHttpRequest.responseXML`, `window.setTimeout`) and callbacks. JSP invocations are reconstructed by recognizing certain calls that occur within the JSP engine, which works well for most applications, although it fails to scale up to bigger applications with multiple JSP files with the same name, but in different directories. A possible solution would be to instrument JSP files prior to analysis, which has the additional benefit of not depending on implementation details of the JSP engine.

3.2 Technical Challenges

Anonymous Functions. One caveat regarding JavaScript tracing is that the language allows a developer to define anonymous functions, a mechanism which is commonly used by web developers. Because many trace visualizations (including ours) display the names of invoked functions, this becomes a problem: e.g., a call tree showing "anonymous" functions calling each other is not particularly helpful. In practice, it turns out that a function is often assigned to exactly one variable, e.g.: `var f = function(...){...}`. Therefore, whenever this is the case, we use the name of the variable to identify the function. We parse all JavaScript files and for every anonymous function definition that we encounter, we try to find a variable or instance variable that precedes it. Note that this approach is not always correct: in the example, f could be reassigned another function. However, the approach seems to work well in practice: for example, the popular Firefox FireBug add-on[4] uses a similar technique (albeit simpler, based on regular expressions) to "name" anonymous functions.

Lazy Loading. Another potential issue is the "lazy loading" of JavaScript files, a technique that is used in the Dojo library, for example. "Lazy loading" refers to retrieving a script file by means of an Ajax request, and subsequently "eval"-ing it, reducing the initial page load time. However, because of the "eval" call, the link between the original filename and code is lost. This can lead to the undesirable situation of having a fragment of code and not knowing where it came from, except that it was dynamically generated at some point. The tool solves this problem by computing a hash code for the response text of every Ajax request, and every "eval"-ed string. When the tool shows a fragment of "eval"-ed code and finds a matching Ajax response text hash, the tool can reconstruct the filename of the "eval"-ed code.

[2] See `http://dojotoolkit.org/`.

[3] Also see `https://developer.mozilla.org/en/Extensions`.

[4] FireBug 1.5.0, see `http://getfirebug.com/`.

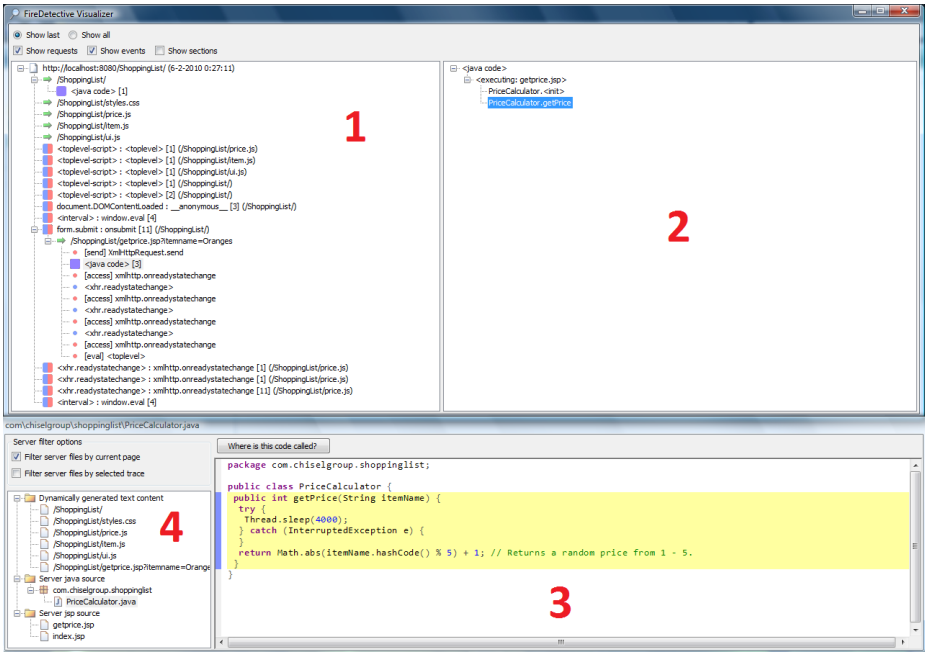

Figure 2: The visualizer, showing an analysis of a small sample application. 1. High-level view. An Ajax request is expanded; related traces/calls are shown. 2. Trace view. 3 Code view. 4. Resource list, showing only the files that were used on the current page.

4. USING FIREDETECTIVE

Goals and Experiences. With FireDetective we have created a tool in which trace analysis is applied to the domain of Ajax web applications. The tool design demonstrates how to employ abstractions from the Ajax/web domain to link execution traces, at both the client (browser) and server-side.

Our goal with FireDetective is to help improve the program understanding process of web developers. In [5] we describe a pre-experimental pretest/posttest user study of which the main result is that participants found that FireDetective helps them to understand Ajax applications (1) more effectively, (2) more efficiently and (3) with more confidence.

Downloadable and Open-Source. FireDetective's homepage is at: `http://swerl.tudelft.nl/bin/view/Main/FireDetective`; the tool is downloadable and contains detailed instructions on how to use it. We tried it out on Windows Vista SP2, with Firefox 3.5.11, Eclipse 3.6 and Java EE 5 with a GlassFish 2.1 webserver. The source-distribution is also available and a video can be found at: `http://www.youtube.com/watch?v=Trp82FNBeEU`

Future Developments. During development and evaluation, the following directions for future developments were identified:

- While tracing small applications happens without significant performance degradation (e.g., the Java Pet Store [5]), the JavaScript tracing should be improved as to allow the tracing of larger applications, without the user being confronted with slowdowns.
- Currently, FireBug and FireDetective cannot coexist as Firefox plug-ins. A number of users pointed toward the fact that it would be helpful to have a feature that allows to inspect variables, something that FireBug offers, but FireDetective currently lacks. Therefore, we plan to look into integrating FireBug and FireDetective.

[5] `https://blueprints.dev.java.net/petstore/`

5. REFERENCES

[1] Bas Cornelissen, Leon Moonen, and Andy Zaidman. An assessment methodology for trace reduction techniques. In *Int'l Conf. Softw. Maintenance*, pages 107–116. IEEE, 2008.

[2] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[3] Jesse J. Garrett. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, retrieved on December 7th, 2010.

[4] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 349–359. ACM, 2002.

[5] Nick Matthijssen, Andy Zaidman, Margaret-Anne D. Storey, Ian Bull, and Arie van Deursen. Connecting traces: Understanding client-server interactions in Ajax applications. In *Int'l Conf. on Program Comprehension (ICPC)*, pages 216–225. IEEE, 2010.

[6] Ali Mesbah and Arie van Deursen. A component- and push-based architectural style for ajax applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.

[7] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Symposium on Visual Languages (VL)*, pages 336–343. IEEE, 1996.

[8] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[9] Andy Zaidman, Bart Du Bois, and Serge Demeyer. How webmining and coupling metrics improve early program compehension. In *Proc. Int'l Conf. on Program Comprehension (ICPC)*, pages 74–78. IEEE, 2006.

TUD-SERG-2011-002
ISSN 1872-5392