Crossfire - Multiprocess, Cross-Browser, Open-Web Debugging Protocol

Michael G. Collins

IBM Research - Almaden

mgcollin@us.ibm.com

John J. Barton

IBM Research - Almaden

bartonjj@us.ibm.com

Abstract

We present *Crossfire*, a system and protocol designed to enable debugging of Web pages in another process or machine. Issues specific to any one Web browser are abstracted by the protocol and implementation, allowing a new generation of Open Web development tools to be implemented. We discuss the major refactoring of Firebug, the open source Web debugging tool to use *Crossfire* and the interplay between goals and resources that such an effort requires. In addition to the cross-browser focus of the protocol, we also discuss support for extensions which themselves will be cross-browser and client-server.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging debugging aids, distributed debugging

General Terms Experimentation, Reliability

Keywords Source-Level Debugging, Distributed Debugging, Open Source

1. Introduction

Web Applications continue to grow in size and complexity. The appearance of AJAX, which enabled data to be downloaded by a Web page in the background, started the surge. This led to the emergence of common toolkits and libraries for JavaScript, which drove performance increases in Web Browsers, fueling more growth in client-side Web application development. These improvements, combined with new features available in Web browsers shifted investment from server- to client-side. Recent empirical analysis of representative major Web sites shows program sizes in the range of hundreds of kilobytes of sophisticated code[22].

To develop and maintain these large applications, programmers and designers rely on numerous tools, most notably Web page debuggers. This paper describes *Crossfire*, a protocol and implementation to provide a next generation platform for Web page debuggers: support for cross-browser, remote and mobile development tools. We describe the major re-architecting of the most widely used Web page debugger, Firebug, to use *Crossfire* to support its client-server communications. Our description focuses on practical, state-of-the-art issues in an on-going, fast-moving, open-source project. Thus we cover details of protocol important for implementation and issues of matching resources to goals important for project management: we must deal with both low and high level issues to be successful.

2. Background

To understand the importance and challenges of the *Crossfire* work we start by introducing Firebug. Released by Joe Hewitt in 2006, Firebug was the first integrated Web debugger. Firebug is a runtime debugger: it directly accesses, responds to, and operates on the running Web browser. Rather than separate views of JavaScript, CSS, and HTML, Firebug integrated its views such that interaction with, for example, an HTML element would cause synchronized views of the CSS rules. Rather than static views of browser state, Firebug included dynamics like network traffic analysis and console logging; rather than read-only views, Firebug allowed live edits where possible so developers could try out changes. The resulting tool became very popular with developers and contributing significantly to the growth in Web applications.

The primary implementation of Firebug is a Firefox extension, a supplemental software component that loads into the Firefox Web browser. A secondary implementation with fewer features and, in particular, limited support for JavaScript debugging, called Firebug Lite, works in multiple browsers. The success of Firebug triggered competitive implementations of Web Page debuggers in other browsers, including DragonFly for Opera[15], Web Inspector for Google Chrome and Apple Safari[5], and the developer tools in Microsoft Internet Explorer[12]. Since 2007 Firebug has been

developed as an open source project, with seven major releases.

To give a flavor of the kinds of operations Firebug supports, we outline an example more completely described in Ref.[17]. Suppose a developer wants to understand why a block of text in the Web page turned green while the page was loading. They might use the Firebug "inspect" feature, moving their mouse cursor over the green text, causing Firebug to display the corresponding HTML element. They see that the element has a *style* attribute setting the color green. While hovering over the green block of text, the developer clicks down to lock the user interface on the element, then moves to the HTML panel in Firebug and right-clicks on the selected HTML element representation. A menu pops up allowing the developer to select "Break On Attribute Change". When the developer reloads the page, Firebug halts in the JavaScript code panel, on the line where the attribute is changed from red to green.

This example illustrates that we will need to synchronize mouse events on the Web page with the debugger UI, identify HTML element representations rendered in the debugger UI with the elements in the Web page and synchronize DOM mutation event handling wit JavaScript execution. Crossfire is designed to support the kinds of features available in current built-in development tools such as Firebug, while enabling the advantages of a remote debug connection.

3. Design Motivation

Crossfire has three main design goals: multiprocess support, remote and mobile debug, and cross-browser debugging. These closely related goals arose out of an interplay between user benefit and development costs. As an open source project we must work with development resources motivated by goals: no matter how much value Firebug users may receive from a goal, the selection must be limited by the motivation of open source contributors.

Necessity motivated the first *Crossfire* design goal, multiprocess support. Soon after the Google Chrome browser was released, the Firefox team at Mozilla began plans to convert Firefox to a multi-process design. The Google browser uses one controlling process for the application and one process for each Web page. This allows the browser to use the operating system isolation to prevent problems on one page from bringing down the entire application and it allows each page to use a different physical processor on modern multi-core computers [11]. Depending upon the Firefox browser platform changes, a shift to multiprocess could render a single-process Firebug debugger unusable.

As a practical matter we could not wait for the new platform to become available: with only two full-time developers plus a number of dedicated but part-time contributors, and a commitment to continuous compatibility with Firefox we had to begin work immediately to ensure that our small resource could complete the transition in time to remain a viable project.

Therefore we assumed that Firefox would eventually adopt an architecture similar to Google Chrome: a client/server split debugger with a back-end in one process and a front-end in another process. We believe that this assumption is planning for the worst case: converting Firebug to client/server is a multi-person-year effort but likely to work with what ever the Firefox team decides to do.

While necessity forced our action, opportunity followed. The client/server choice, if successful, adds two new dimensions to Firebug for users: remote debug and mobile device debug. We expect the value of these dimensions to grow as more developers work in distributed teams and as mobile plays an increasingly important role in Web application development. In fact this value was recognized by the DragonFly Web debugger for Opera[15] well before the Google Chrome browser.[5] The additional cost of designing for remote and mobile debug on top of a client/server design – primarily mechanisms for specifying the connection addresses – comes with potentially high benefits. Moreover, the benefits align with directions important to the project's primary open source contributors (IBM and Mozilla).

The final goal of cross-browser debugging offers even more benefits to Firebug users. Web application developers by definition target all Web users, but not all Web users are running identical Web platforms. Almost all potential users of a Web site will be running one of few similar but slightly different browsers. The commonality allows Web developers to do most of their work on one browser, then test for differences on other browsers. Of course in the latter case they need to debug the problem on a browser with unfamiliar debugging tools. A common debugging tool across the major browsers would help with this common and significant problem.

The benefit of cross-browser debugging comes at a high cost for the project. Instead of one server and one client, we face at minimum one server for every browser. And for each server we have to deal with both the slight differences in browser implementation of standard Web APIs and potential large differences in how debuggers can connect to the browser.

Unlike commercial or pure research projects, a community-driven open source project like Firebug might balance the cost of implementing cross-browser debugging support by attracting more contributors interested in this particular goal. That is, by adding this costly goal we can attract new contributors, allowing us to create more total value. In particular new contributors from the Orion project[16], joined to create a *Crossfire* server for Microsoft Internet Explorer and from the Eclipse project[10] to create a new *Crossfire* client in Java for connecting to Eclipse. Moreover, a cross-browser client for Web debugging can be largely implemented with

Firebug Lite code, allowing our project to consolidate developer resources around fewer lines of code to maintain.

Our three design goals created constraints for the *Crossfire* implementation. Above we outlined how the multiprocess support lead to a client-server design choice. Support for remote and mobile debug forces isolation of user interface to the client (excepting some small interface for connection specification). The cross-browser goal creates constraints indirectly: to minimize the extra cost of supporting multiple servers we chose to adopt the Google Chrome communications channel (sockets) and wire protocol format (JSON). Neither Firefox nor Internet Explorer had existing servers, so they did not alter our choices. Opera had a server but no one on our open source team planned to work with Opera and the server itself was not open source making implementation more difficult. Since Firebug is already written in JavaScript, JSON format is especially easy to work with and has good performance[20]. For the communications protocol, HTTP would be a better choice for the project: the JavaScript support for HTTP is much better than sockets and HTTP works better in practical remote scenarios through firewalls. However we made the judgment that better socket support was coming in the near future[1], support was adequate now, and lowering cost on a Google Chrome back-end was important. In addition our goals imply that the client and the communication protocol should be built from open Web standards to maximize the reuse across target browsers.

4. Evolution not Revolution

In addition to motivating developers to contribute to *Crossfire*, we also need to motivate users to help us test and refine the system. As a practical project supporting 3 million users, Firebug provides a large base of experienced Web developers working with a broad spectrum of Web technologies. An open source, working, state-of-the-art debugger motivates users to explain problems, create test cases, provide documentation and to help other users with problems that come up. To harness this unusual resource for *Crossfire* we need a plan for incremental refactoring of Firebug to be compatible with *Crossfire*. The refactoring plan needs to provide waypoints for the development and it needs to provide intermediate value to users and/or contributors.

4.1 Inter-application JavaScript Debugging

The first intermediate state for *Crossfire* is shown in Fig.1. In Firefox we implemented a *Crossfire* server limited to support for JavaScript debugging. In Eclipse we implemented a *Crossfire* client. This allows the user interface in Eclipse to control and examine the JavaScript program running in Firefox. A proprietary version of the client shipped in IBM's Rational Application Developer product for two years, then the open source Eclipse team created a new implementation as part of it's JavaScript Developer Tools (JSDT) project[10]. By working towards the Eclipse team's goals of remote
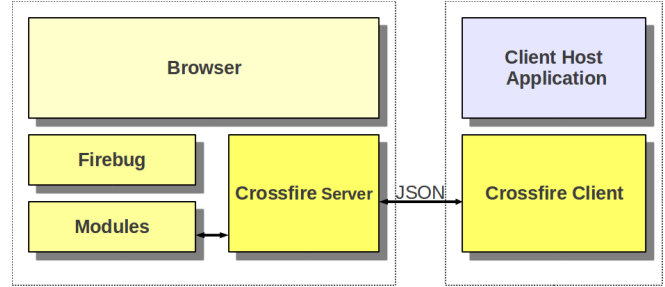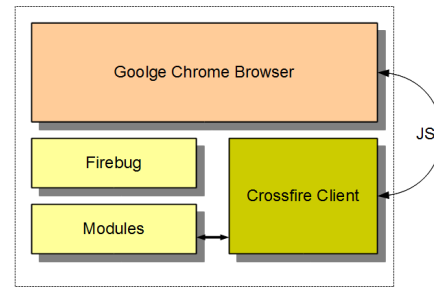

Figure 1. Inter-application JavaScript Debugging *Crossfire* architecture. *Crossfire* versions 0.1 to 0.3 connected to an Eclipse plugin, supporting simple JavaScript debugging

JavaScript debugging, this stage of the work provided valuable implementation experience and engagement with the Eclipse team. This work has been released to users but continues to be improved.

4.2 Intra-browser JavaScript Debugging

The second intermediate state implements the client side of the JavaScript part of *Crossfire* in a Web browser as sketched in Fig. 2. While this diagram seems a bit bizarre, with the

Figure 2. Intra-browser JavaScript Debugging *Crossfire* architecture. *Crossfire* versions 0.4 targets supporting simple JavaScript debugging with the client and server in the same application.

debugger running in and connecting back into the the same browser, this step allows us to add JavaScript debug support to the Firebug Lite implementation in the Google Chrome browser while we simultaneously refactor the Firefox *Crossfire* server to resemble the Google Chrome back end.

The key reason this architecture makes sense is that a large part of the non-JavaScript parts of a Web page debugger uses standard Web APIs. That means that three different applications, Firebug Lite running as co-resident with a Web page, Firebug Lite running as a Google Chrome extension, and the HTML/CSS/Console debug support code in Firebug for Firefox can use identical code in different wrappers. By re-engineering our current somewhat divergent code to group the identical parts we reduce maintainence. By adding JavaScript debugging using the now platform-independent Firefox code to the Firebug for Google Chrome code we add

user value: the beginnings of cross browser development. Both efforts contribute to our final goals.

Furthermore, the two JavaScript-only *Crossfire* servers, one for Firefox and one for Google Chrome, will be able to support alternative clients. In particular, the Orion project, a Web based Web-development system, plans to support JavaScript debugging over *Crossfire* on their editor user interface. In return that project is implementing a *Crossfire* server for Internet Explorer, allowing us to cover more than 75% of the browser market with *Crossfire* supported tools. Versions with these features are scheduled to complete in June, 2011.

4.3 Cross-browser Debugging

The final stage completes the transformation of an in-process single-browser Web page debugger to a client-server cross-browser tool as shown in Fig. 3. Conceptually we simply re-
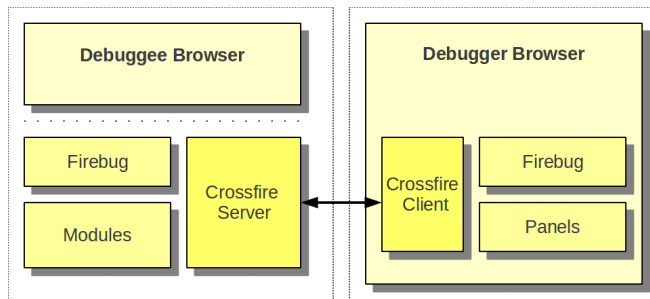
Figure 3. Cross-browser Debugging architecture, proposed.

apply the approach ironed out in the previous step to the rest of the program and arrive at the complete value proposed at the outset. In practice, the concept hides a lot of work. Many lines of code must be carefully divide into two piles and the whole must be made to work again. This work is scheduled to complete in Dec. 2011.

4.4 Modules and Tools Interface

In parallel with the architectural changes outlined above, we also need to make important infrastructure improvements. Two such improvements are of particular interest: conversion of the source code to 'modules' and introduction of a cross-browser JavaScript tools application programming interface (API).

Modules   The original Firebug for Firefox and Firebug Lite code used HTML `<script>` tags to load and compile the source. This approach has two major drawbacks: 1) all of the top-level symbols in each file mix with the top-level symbols of any other files loaded in the same scope and 2) the load/compile steps are serialized. The first drawback never affected Firebug code because all of the files encapsulated their symbols in function scope. But the second one means that loading code always causes an upfront overhead to starting the application.

A replacement for `<script>` tag loading will need to support both client and server sides of a refactored Firebug and it must allow us to debug our own code. As in other cases, we also want a solution that avoids additional work by the development team. The solution we adopted was *RequireJS*[18], a form of a module loader inspired by the CommonJS[14] open standards effort. For the client side or Firebug Lite we can use this loader directly once we change our source to its format. For the server side we needed to implement code to read source files within the Firefox platform and to support debugging.

Tools Interface   During the transformation from monolithic to client-server application, Firebug needs to operate in both modes. The obvious way to deal with this is to introduce a programming interface between the front and back ends. In Firebug for Firefox, the interface functions call the back end directly; in the intermediate and cross-browser version these functions call the *Crossfire* client API. Similarly in the back end, the multiprocess versions of the interface call the *Crossfire* server. Notice that this interface becomes a natural programming layer for interaction between parts of the debugger. Since *Crossfire* is designed to be cross-browser, with some care in the design, the programming interface we create becomes a general purpose Tools Interface for Open Web development.

The module and Tools Interface infrastructures complement one another. Each logical chunk of the Tools Interface corresponds to the exported symbols from one of the modules and the conditional assembly of the application from modules, to become either in-process or client-server, works by using the module loader to select the appropriate implementations of the interface.

5.   Crossfire

The Crossfire protocol is an asynchronous, bi-directional protocol designed to enable the full functionality of the Firebug debugger in a multi-process or remote scenario. Where it was possible, the design of the protocol took cues from existing debug protocols such as DBGP[19], Opera Scope[8], Google's Chrome Dev Tools[4] , as well as common Web technologies (e.g. HTTP, JSON[20]). Certain features unique to Firebug and to debugging code running inside a Web Browser had to be taken into account in the design of the protocol. We give an overview of the protocol and discuss some aspects that are important to its design.

5.1   Overview

Debugging the code that implements a Web page or Web application differs in some significant ways from debugging applications developed for other types of systems. HTML and CSS are used to declaratively specify the structure and style of the user-interface, which is rendered by the Web browser. Developers cannot (easily) debug the rendering code itself. Instead, built-in tools like Firebug allow the

developers to interact with the rendering engine by modifying the input and observing the output in real time, via live CSS and DOM editing. JavaScript code on the page can be stepped through when it is executing, however there is no guarantee that any JavaScript code is necessarily running at any given moment. JavaScript code on a Web page can be triggered by timers, user interactions or network events. There is no outermost main or idle loop to return to; when a section of JavaScript code is finished executing, control returns to the Web browser. This confounds attempts at things such as a simple 'halt' command, which is common among debuggers for other systems. The closest analog in Firebug is the *break-on-next* feature.

The Crossfire protocol is heavily event-driven, and requests made via Crossfire are asynchronous. This differs from many other debug protocols which are often synchronous or a combination of synchronous and asynchronous calls. The reasons for this decision stem from the nature of code running in a web page, and the fact that in some scenarios, especially the intermediate scenarios we wished to achieve, we would have a remote client connected to a server which also had a co-resident debug UI (Firebug). In other words, a complete Firebug and Crossfire server implementation would be running in a single Firefox process, and clients could connect to the Crossfire server. The result of this scenario is that clients cannot assume or rely on being the only agent acting on the runtime engine. For instance, a Crossfire client cannot safely assume that the debugger will remain suspended on a line of code until the client issues a request to resume. It is also possible that this action was triggered by the user from another client (in this case it is helpful to think of Firebug's in-process UI as another client to the debugger). However any connected Crossfire client will receive an event whenever the JavaScript debugger suspends or resumes, and should react accordingly.

Implementations of the protocol differ based on whether the implementation is intended to operate as a client or server. A Crossfire server resides in or is connected to the process which is acting as the runtime platform for the Web page, application, or other code which is to be debugged. This is typically a Web Browser, although supporting other runtime environments is envisioned. A Crossfire client connects to a server in order to receive events and issue requests, typically in order to provide a user-interface for debugging, (e.g. GUI or command-line debugger). It is not necessary for the client and server to reside in the same process or even the same host machine.

5.2 Connection and Handshake

To avoid conflicts with existing ports, Crossfire does not specify a standard or well-known port. Port agreement is left up to the user, or the client software must start the server listening on the same port it will attempt to connect to.

The connection protocol is purposefully conventional. The Crossfire server listens for a TCP connection on the specified port (greater than 1024). A client wishing to connect sends the string "CrossfireHandshake" followed by a CRLF (a blank line specified by a carriage-return followed by a line-feed character, as with HTTP). An optional second handshake line may contain a comma-separated list of tool names to be enabled immediately, followed by another CRLF. The server replies with the same handshake string, at which point the connection is established and the client may begin sending requests and receiving events from the server.

5.3 Client/Server Behavior

Once a connection has been established and a successful handshake is completed, the server may begin sending events to the connected client using the same TCP connection used for the handshake. A client may also begin sending requests to the server using the same connection. Clients should not expect the server to respond synchronously to requests or in order. The most common example is a Crossfire server sending one or more events to the client before responding to a client's request, because the events occurred during the time the request was being sent or processed.

5.4 Message Packets

As described above, the packet format follows the design of the Google Chrome browser[13]. A well-formed Crossfire packet contains one or more headers consisting of the header name, followed by a colon (":"), the header value, and terminated by a CRLF. A "Content-Length" header containing the number of characters in the message body is required, and additional headers are allowed.

The message body is separated from the headers by another CRLF blank line. The blank line is followed by a well-formed JSON string. The message must contain a "type" field with the value one of "request", "response", or "event", and a "seq" field which contains the sequence number of the packet. The sequence number of each message should be greater than that of the last message received.

5.5 Contexts

Unlike desktop or server application debuggers, a Web browser typically runs multiple applications or Web pages. A developer is likely to debug one or two of these applications, while the rest are unrelated to the application. The debugger must have a mechanism to focus on the particular page being debugged. Firebug represents an instance of a Web page via an object called a *Context*. The context object allows Firebug's panels and modules to share information about a web page that is being debugged, therefore it has a central role in Firebug's architecture.

Most of the events that occur in a Web browser that are of interest to a debug UI are related to individual pages, and therefore individual Firebug contexts. Examples of context-specific events include loading (or reloading) of a page, loading and compiling a script, errors being generated from

an executing script, DOM elements being added or modified, a breakpoint being added to a script, etc.

The Crossfire protocol uses contexts for most requests / events. Crossfire represents a context as a mapping of the unique context ID and the URL of the page. This allows a connected Crossfire client to distinguish between separate loads of the same URL, as is often the case when a developer reloads a page several times in the course of developing or debugging the page. Firebug's TabWatcher component monitors loading and unloading of Firefox windows and tabs. The Crossfire server assigns a unique identifier to each context, and passes this ID as part of most event and response packets.

5.6 Breakpoints

Breakpoint debugging is a standard tool for debugging software at runtime in many languages and environments. The Web Browser environment creates several challenges for designing a remote protocol which supports breakpoint debugging. Firebug also introduces several types of breakpoints which are not present in other environments [17].

Even the simplest case, a JavaScript line breakpoint, has design implications that must be considered. Typically, such a breakpoint is identified by a line number and the URL of the script. However, existing JavaScript debugging APIs such as Firefox's do not contain the concept of Firebug's contexts. Therefore if a user places a breakpoint on line 23 of a URL http://localhost/script.js, then that breakpoint will exist for all occurrences of that URL. This may or may not be what the user actually desires. Considering the increasing use of JavaScript libraries, it is entirely likely that a script from the same URL is loaded into two completely unrelated pages. It is conceivable that a user would wish to debug his or her code and the interaction with the library, without affecting code running in another tab or window.

Crossfire's breakpoint protocol allows breakpoints to be set in one of two ways, either with or without a context ID. If a Crossfire client specifies a context ID along with a request to set a breakpoint, then that breakpoint should be enabled for that location in any existing context, or a future context which is created with the same URL in the same container (i.e. the page is reloaded).

If a client does not specify a context (by passing null as the value of the context ID), then the behavior is to set a breakpoint for the specified location in any future contexts. The intended use case for this behavior is setting a breakpoint in a client UI such as an editor, where the source code location has changed (due to editing), but the changes have not yet been applied to the page in the browser.

More advanced breakpoints, such as the HTML element breakpoints, are supported by specifying that the *location* property of a breakpoint in Crossfire is an arbitrary JSON object. A location object is defined depending on the type of breakpoint. A JavaScript line breakpoint has a location object which consists of a target URL and line number.

Firebug's HTML breakpoints have a location object which consists of an XPath expression that identifies the target element. Eventually other location types may be added, e.g. to support network-related breakpoints such as Firebug's *BreakOnXHR* feature.

5.7 Extensibility

One of the goals of Crossfire is to support remote and multi-process versions of Firebug. One of Firebug's features is its ability to be extended, and there are already many existing extensions. Therefore, we have developed what an API for Crossfire, called the *Crossfire Tools API*. The Tools API allows Firebug extension developers a clean and consistent way to access the Crossfire client and/or server connection.

On the server-side, the Tools API allows an extension to send custom events and handle custom requests using Crossfire's connection and transport mechanism. A client extension can listen for these events and respond to the requests. Using this API, it will be possible for Firebug extensions to continue to adapt to architectural changes in future versions of Firebug.

One consequence of this design choice is that the set of possible commands or event names cannot be specified definitively by the protocol. A Crossfire client or server must therefore be able to accept and respond to any well-formed message packet, even if it may not know how to handle a particular command or event type.

6. Implementation

6.1 Crossfire Firefox Extension

The first implementation of the Crossfire protocol is an extension to Firefox and Firebug. Implemented entirely in JavaScript, it uses a modular design to allow us to share code between client and server implementations, as well as cross-browser implementations. The transport layer operates as either a Crossfire client or server; currently the transport operates over TCP sockets. Future support for operating over HTTP and/or WebSockets is planned.

The extension can be started in client or server mode either from the Firefox user interface, or via command-line switches to Firefox. This latter mode of operation allows external tools to launch Firefox and start the Crossfire server listening on a known port so that the external tool may automatically connect back to it.

6.2 Crossfire Tools API

To support extensions, *Crossfire* maintains a registry on both the client and server side as show in Fig. 4. The Crossfire extension also implements an API, called the *Crossfire Tools API* which enables extensibility of the Crossfire system and protocol. Firebug features such as the Console, Inspector, and Net Panel, are implemented as tools using the API, allowing them to be enabled/disabled independently.

A tool can be implemented as a JavaScript file or collection of files that implements the Crossfire Tools API. The tool registers itself with the core Crossfire Module, providing an identification string that is used to identify messages via the 'tool' header. In server operation, the tool is then able to receive notification when a connection is created or when request packets are received. The Tools API allows a tool to access Crossfire's transport layer in order to send events or command responses. Typically, a tool operating within the context of a Crossfire server might register listeners with one or more Firebug modules, in order to dispatch events generated by the module to the remote connection. A tool operating as part of a Crossfire client would process the events sent from the server tool, and update part of the client UI, such as a Firebug panel. The tool could also listen for client events from the UI, and send the appropriate requests to the server, to be handled by the tool's server-side component.
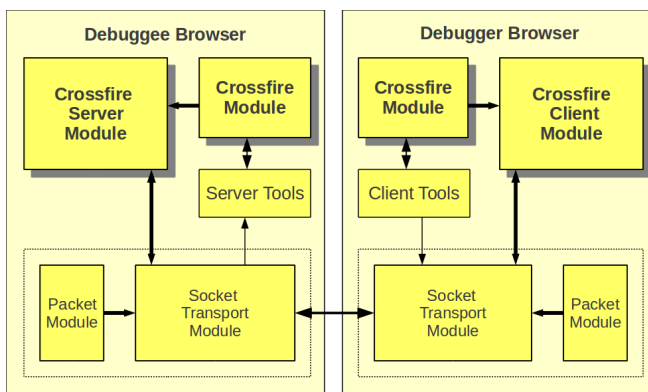
Figure 4. Crossfire Extension Architecture. The Crossfire Module running in both client and server load extension modules which register 'tools', one half in the Server Tools registry and the other half in the Client Tools registry.

7. Related Work

7.1 Multiprocess Web Debug Tools

As we discussed above, Opera's DragonFly[15] implements a complete remote debug solution and Google Chrome's Web Inspector works across processes. These implementations are specific to their host browser.

7.1.1 Weinre

The Weinre (Web Inspector Remote)[21] project implements a partial Web page debugger (no JavaScript debugging) by adding JavaScript code to a Web page in a proxy server much like Firebug Lite. The added code connects back to the proxy which then re-transmits to a third process running the user interface code from WebKit's Web Inspector. The main target for this work is mobile devices. If a *Crossfire* server were implemented in Weinre, the proxy could support connections to *Crossfire* clients.

7.1.2 Eclipse JSDT

The Eclipse JavaScript Development Tools (JSDT) project[10] includes a *Crossfire* client implementation (currently in incubation). This code is written in Java and supports connections to Firebug's server as well as an early implementation of *Crossfire* in Internet Explorer.

7.1.3 Orion

The Orion project[16] aims to create Web development tooling based on Web technologies, and plans to use Firebug and Crossfire as part of their debugging support. Discussions with the Orion team helped inform the design of *Crossfire*.

7.1.4 Cloud 9 IDE

The Cloud 9 IDE[6] supports remote JavaScript debugging (only) from a Web page to Google's V8 engine running in a `Node.js` server.

7.2 Remote Protocols

Many protocols have already been designed for the purposes of remotely debugging an application running in another process, virtual machine, host, etc. The GNU GDB debugger has an associated Remote Serial Protocol (RSP)[7]. While it is the only debugger we know of with its own song [3], it is primarily designed for debugging native code, particularly on embedded systems, and would not be well-suited for use with Firebug. The Java Debug Wire Protocol (JDWP) [2] provides remote debugging of Java Virtual Machines. The protocol supports command and response message pairs similar to Crossfire and other Web debugging protocols. However the design of the protocol, particularly the synchronous aspects, would not work well with Firebug's existing architecture.

7.2.1 DBGp

DBGp[19], is an acronym for Debug Protocol, and was developed for version 2 of the XDebug debugger for the PHP language. Although it was designed not to be language specific, many of the commands are intended to be synchronous, as opposed to the asynchronous nature of Crossfire. DBGp also allows for the debugger engine to send an event to a client via the 'notification' element, with a custom body. However in order to support Firebug, we would need to define the same events defined by Crossfire as DBGp notifications, essentially creating another protocol within the protocol.

7.2.2 Opera Scope Protocol

The Opera browser has a built-in Web development tool called DragonFly, which also supports the Scope remote debug protocol. The Scope protocol[8] supports XML and JSON formats, and features such as JavaScript debugging and remote DOM inspection. It is used to allow the desktop DragonFly client to connect to another Opera browser instance, including mobile versions.

7.2.3 V8 / Chrome Dev Tools Protocol

The V8 protocol[13] is a JSON-based wire protocol for debugging JavaScript programs running within the V8 engine. The Chrome Dev Tools protocol[4] wraps the V8 protocol to provide the additional information needed to debug a Web page running within Google's Chrome browser. The protocol implements JSON messages over TCP/IP sockets, and was the basis for much of the initial work on the Crossfire protocol. However, over the course of developing Crossfire and refactoring of Firebug, it was realized that Crossfire would require more functionality than the Chrome protocols provided.

8. Future Work

8.1 Web Sockets

Implementations of the WebSocket API[1] and protocol[9] standard are beginning to appear in recent versions of several Web Browsers. The WebSocket API allows JavaScript code in a Web page to create a full-duplex socket connection to another host using a lightweight protocol. Using these API's, it may be possible in the future to provide a Firebug front-end (similar to Firebug Lite) and Crossfire client which do not rely on other browser-specific extension APIs.

8.2 Mobile Web Debugging

Mobile devices such as smart phones and tablet computers now include full-featured Web browsers. In contrast to Desktop browsers, which have been adding more tools for developers, mobile browsers do not have the built-in Web development tools. The intuitive reason for this lack of tools is that the form factors of these mobile devices do not lend themselves to software development tasks. In this scenario, the remote debugging solution may be the only viable alternative.

8.3 Multi-user Debugging

The architecture and system we have built thus far has been implemented and demonstrated with a single user in mind, but is not restricted to that. Even in cases with a single user, there could be cases where it is desirable for a Crossfire server to support connections to multiple clients, such has connecting an external IDE while also using Firebug's in-browser UI.

Though the default operating mode of our Crossfire server implementation is to accept incoming connections only on the local host interface, it is possible to connect from a remote host. Since Crossfire should support multiple clients, it is conceivable that multiple users could use separate Crossfire clients connected to the same Web page instance to collaborate on developing or debugging that page. While it is beyond the original goals of the Crossfire project, it would be possible to build on the Crossfire work to add additional features to facilitate this kind of collaborative debugging.

9. Conclusion

Our work thus far has demonstrated that it is possible to incrementally refactor and rearchitect an existing codebase while maintaining the ability to support Firebug's large userbase with releases which are compatible with new releases of Firefox. In addition we have shown it is possible to implement a system for remote debugging similar to existing solutions in other browsers, but using a modular approach that is written purely in JavaScript. The project has been successful in attracting new contributors and new opportunities for Firebug, enabling the project to explore new directions. Continuing this work will provide numerous benefits for Firebug users as well new features that are in demand, while allowing Firebug to adapt to possible future changes in Firefox, and increasing the features offered by Firebug/Firebug Lite in other Web browsers.

Acknowledgments

References

[1] WebSocket API Specification, 2001. `http://dev.w3.org/html5/websockets/`.

[2] *Java Platform Debugger Architecture*, 2004. `http://download.oracle.com/javase/1.5.0/docs/guide/jpda/`.

[3] GDB Song, 2007. `http://www.gnu.org/music/gdb-song.html`.

[4] Google Chrome Dev Tools Protocol, 2009. `http://code.google.com/p/chromedevtools/wiki/ChromeDevToolsProtocol`.

[5] WebKit Web Inspector, 2010. `http://trac.webkit.org/wiki/WebInspector`.

[6] Cloud 9, 2010. `http://cloud9ide.com/`.

[7] *GNU Debugger (GDB) Manual*, 2010. `http://sourceware.org/gdb/current/onlinedocs/gdb/`.

[8] Opera Scope Protocol, 2010. `http://dragonfly.opera.com/app/scope-interface/`.

[9] WebSocket Protocol, 2010. `http://www.whatwg.org/specs/web-socket-protocol/`.

[10] Eclipse JSDT, 2011. `http://wiki.eclipse.org/JSDT/Debug`.

[11] Google Chrome, 2011. `http://www.google.com/chrome`.

[12] Microsoft Internet Explorer Developer Tools, 2011. `http://msdn.microsoft.com/en-us/library/dd565628`.

[13] V8 Debug Protocol, 2011. `http://code.google.com/p/v8/wiki/DebuggerProtocol`.

[14] CommonJS, 2011. `http://www.commonjs.org/`.

[15] Opera DragonFly, 2011. `http://www.opera.com/dragonfly/`.

[16] Orion, 2011. `http://www.eclipse.org/orion/`.

[17] J. J. Barton and J. Odvarko. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 81–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: http://doi.acm.org/10.1145/1772690.1772700. URL `http://doi.acm.org/10.1145/1772690.1772700`.

[18] J. Burke. RequireJS, 2011. `http://requirejs.org/`.

[19] S. Caraveo and D. Rethans. DBGP, A common debugger protocol for languages and debugger UI communication, Draft 16, 2007. `http://www.xdebug.org/docs-dbgp.php`.

[20] D. Crockford. JSON. `http://json.org`.

[21] P. Mueller. Weinre, 2011. `http://pmuellr.github.com/weinre/`.

[22] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: http://doi.acm.org/10.1145/1806596.1806598. URL `http://doi.acm.org/10.1145/1806596.1806598`.