Using Processes to Improve the Reliability of Browser-based Applications

Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy
{creis, bershad, gribble, levy}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington

University of Washington Technical Report UW-CSE-2007-12-01

Abstract

Web content now includes programs that are executed directly within a web browser. Executable content, though, creates new reliability problems for users who rely on the browser to provide program services typical of operating systems. In particular, we find that the runtime environments of current browsers poorly isolate applications from one another. As a result, one web application executing within the browser can interfere with others, whether it be through an explicit failure or the excessive consumption of resources. Our goal is to make the browser a safe environment for running programs by introducing an isolation mechanism that insulates one application from the behavior of another. We show how to use OS processes within the browser to safely isolate programs in a way that is both efficient and backwards compatible with existing web sites.

1 Introduction

The nature of content on the web is changing, revealing weaknesses in the reliability of current web browsers. We are seeing a shift in web content from passive documents to active programs that run directly in the browser, using languages such as JavaScript. Unfortunately, web browsers offer poor isolation between browser-based applications, allowing them to interfere with each other in undesirable ways. For example, a single browser-based program can crash the browser, along with all open browser-based programs. As another example, one CPU-bound browser-based program can block interaction with any other programs in the browser. Finally, a memory leak in one browser-based program can bring down the entire browser.

The goal of our work is to create an isolation mechanism that prevents these problems from occurring. Our solution must satisfy three constraints. First, it must be *safe*, allowing browser-based applications to run side by side, without undesirable interactions. Second, it must be *efficient*, imposing little runtime or space overhead. Third, it must be *backwards compatible*, supporting the millions of web pages on the Internet without requiring changes to content.

Reliability problems in software runtime environments are not new. For example, early PC operating systems, such as MS-DOS or Mac OS, supported only one address space. The result is that programs could easily interfere with each other or crash the operating system. As these systems became more popular and users began to run more programs simultaneously, isolation became a requirement to guarantee reliability.

We propose to use the same mechanism that resolved reliability problems in earlier systems: the process. By placing each browser-based application in a separate OS process, we ensure that it is safely isolated. Separate address spaces ensure that failures in one process do not affect other processes, and that each process can be cleanly terminated. Preemptive scheduling allows multiple processes to run concurrently. Using process-based isolation is efficient, as processes have low startup and context switching times relative to browser rendering times. A greater challenge is maintaining backwards compatibility, as we must ensure that we do not introduce boundaries between web pages that could previously communicate with each other via shared memory. We design an approach that respects known inter-page communication channels in the browser. Specifically, by placing pages from different domains in different processes, we can support effective isolation without breaking existing pages.

We have implemented our safe, efficient, and backwards compatible browser by using a separate OS process in Konqueror for each domain a user visits.

We show that our prototype functions properly on content that causes other browsers to fail, yet continues to function transparently in almost all other cases. In addition, overhead is low.

The Rest of this Paper

This paper is organized as follows. We present measurements of web content in Section 2, showing a substantial rise in the use and complexity of JavaScript-based applications. In Section 3, we demonstrate reliability problems in current web browsers, using a combination of test pages and observed web sites. We propose our solution to these problems in Section 4, leveraging process boundaries between browser-based applications at a granularity that does not break important communication channels. We describe our prototype browser and evaluate its success in solving these problems in Section 5, showing that we achieve safety, efficiency, and backwards compatibility. Finally, we present related work in Section 6 and conclude.

2 Programs on the Web

This section demonstrates the extent to which popular web pages have become active programs rather than simple documents. We focus on applications that use JavaScript because they are widespread and relevant for all web browsers. We show that: *(1)* there has been a dramatic increase in the number of active programs over the past few years, and *(2)* these applications are larger and more complex than before. These programs lead to higher contention for resources in the browser, which increases the importance of providing a reliable runtime environment.

In October of 2006, we measured the amount of JavaScript code on the front pages of the top 100 English language domains, as reported by Alexa [2]. While the Alexa data set may be biased by the user population of the Alexa Toolbar, it offers a reasonable sample of frequently visited sites. To study changes in JavaScript usage over time, we used the Internet Archive [5] to collect the list of the Alexa top 100 domains from the previous three years. We then measured the content for those pages (when available) through the Internet Archive.

We measured JavaScript usage by computing the size of all JavaScript code delivered when visiting each page. This includes the contents of `<script>` tags, externally referenced scripts, event handler code on other HTML tags (*e.g.*, `<a onclick="...">`), and links with "`javascript:`" URLs.

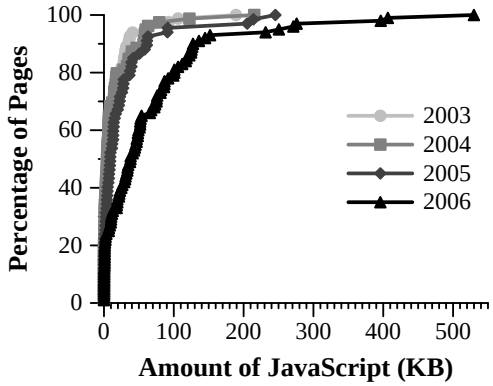

Figure 1: CDF of JavaScript code size for the front pages of the Alexa top 100 domains.

2.1 More Active Content

We first quantify the increasing amount of executable content on the web. We show that over the past few years, many more sites are including larger amounts of JavaScript code.

Figure 1 shows a CDF of JavaScript usage for these pages, over the past several years. The graph shows a dramatic rise in JavaScript code per page, from an average of 15.3 KB per page in 2003 to 64.9 KB per page in 2006. There are now many more pages delivering large amounts of active code. The percentage of pages with at least 10 KB of JavaScript code has risen from 35% in 2003 to 71% in 2006, and 20% of current pages now contain 100 KB of JavaScript code or more. Additionally, the rate of increase of JavaScript code is accelerating. For the 28 pages that appeared in the top 100 list for all four years, the average yearly increase was 1.7 KB in 2004, 17.9 KB in 2005, and 56.1 KB in 2006. This represents a substantial trend toward executable code being delivered to clients' web browsers.

The above results are conservative. Because the data set only includes the front page of each domain, it omits many popular sub-domains (*e.g*, `http://maps.google.com`), many of which may offer specialized applications. For validation, we compare against Netcraft's list of top 100 visited sites [8], which does include sub-domains.[1]

Figure 2 shows the Alexa and Netcraft data sets for 2006. The average amount of JavaScript code per page increases from 64.9 KB to 84.9 KB, with 20% of pages containing over 130 KB of JavaScript. The sites on the Alexa and Netcraft lists significantly

[1]We remove non-English pages and duplicates from the Netcraft list, for an accurate comparison.

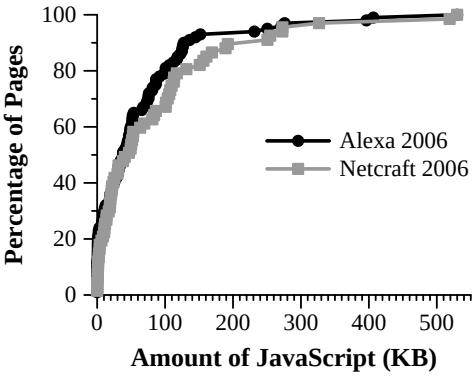

Figure 2: CDF of JavaScript code size for Alexa's and Netcraft's lists of top 100 domains.

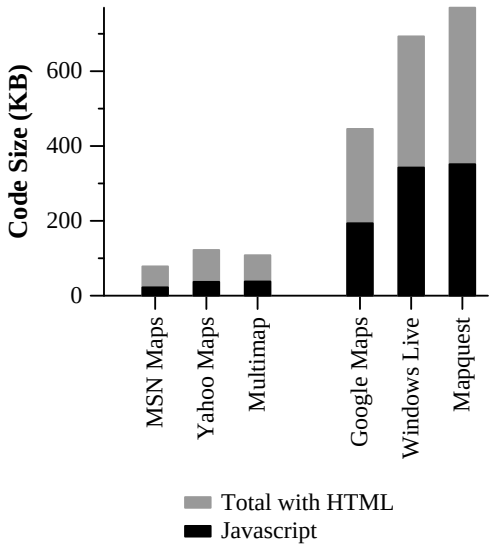

Figure 3: JavaScript code sizes for 3 traditional map sites (left) and 3 interactive map sites (right).

overlap apart from sub-domains, suggesting that the sub-domains are largely responsible for this increase.

2.2 Rich Active Content

In addition to an increase in the number of sites using active code, the available programs are becoming larger and more complex. These programs make use of JavaScript libraries and asynchronous communication, which results in larger programs that increase the workload on the browser.

We find that JavaScript programs are becoming more ambitious, with many attempts to offer the functionality of desktop applications. These include email clients (e.g., Gmail, Zimbra, Zoho), calendars (e.g., Kiko, JotSpot, Zoho), word processors (e.g., Writely, JotSpot, AjaxWrite), spreadsheets (e.g., Google Spreadsheets, NumSum, AjaxXLS), and many others.

Application such as these have given rise to numerous JavaScript frameworks designed to ease development. These include Dojo, Prototype, script.aculo.us, the Yahoo UI Library, Microsoft Atlas, and the Google Web Toolkit, among others. All include libraries of JavaScript code designed to run in the client's browser.

Additionally, these applications provide greater interactivity through the use of asynchronous communication with a server. This allows programs to fetch and display new data without requiring a full-page refresh. To quantify this use of asynchronous communication, we look at how many sites from our Alexa data sets use the `XmlHttpRequest` (XHR) class in JavaScript. Other communication techniques exist as well, so quantifying XHR use provides a lower bound. For 2005, we found only one domain in Alexa's top 100 that used the XHR class on its front page (i.e., Friendster), and we found no uses in the top 100 domains in 2003 or 2004. In contrast, 28 out of Alexa's top 100 domains used the XHR class in 2006, indicating a sharp increase in adoption over the past year.

To see the effect of these trends, we note that *(1)* JavaScript programs are getting bigger and *(2)* these programs increase the CPU workload for the browser. We show this by comparing the code sizes and computational requirements for a sample of traditional and interactive JavaScript applications. We focus on three traditional map services (MSN Maps, Classic Yahoo Maps, and Multimap) and three interactive map services (Google Maps, Windows Live Local, and Mapquest), as of October, 2006. The interactive map sites use asynchronous communication to present maps that can be dragged or zoomed without requiring a full-page refresh, unlike traditional map sites. These interactive sites have only appeared recently: both Google Maps and Windows Live Local were introduced in 2005, and Mapquest transitioned from a traditional site to an interactive site in late 2006.

Figure 3 shows the amount of code distributed with each map site when it is opened to display the 98195 zip code. There is an average of 32.3 KB of code for traditional sites and 296 KB for interactive sites, a difference of nearly an order of magnitude.

Figure 4 shows the CPU time consumed by the sites for both the initial page load and a set of simple operations, including zooming in twice and moving north once. The CPU time is a combination of user time and system time, as reported by the Unix

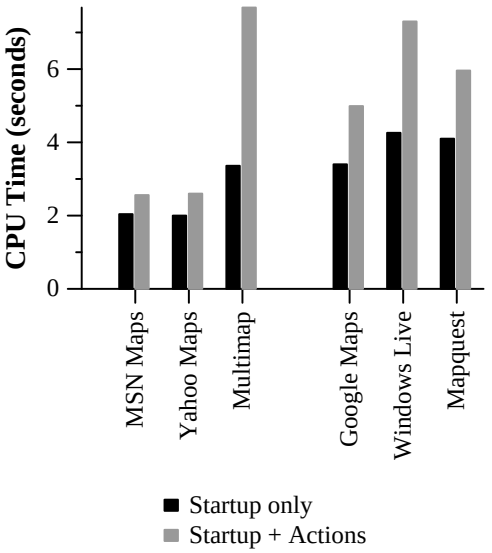

Figure 4: Computation time for 3 traditional map sites (left) and 3 interactive map sites (right). The black bars represent the CPU time required to load the site, and the gray bars represent the CPU time to load the site, zoom in twice, and move north once.

`time` utility.[2] The interactive sites use much more CPU time than most of the traditional sites, for both loading the page and performing simple operations. The exception is Multimap, a traditional site with high CPU utilization. While Multimap does not provide an interactive, draggable map, it does have a large number of CPU-intensive advertisements, and it loads 10 separate frames and 48 KB of JavaScript.

2.3 Summary

From a standpoint of reliability, these increases in code size and complexity for browser-based programs indicate that browsers need to effectively handle resource contention. Increases in program execution time show the importance of running these programs concurrently. Larger and more complex programs present heavier memory contention, so support for independent memory management is crucial. Finally, as users run more of these ambitious applications in the browser, program crashes can be critical without proper failure isolation.

[2]Numbers were collected in Firefox 1.5 on Linux on a 3.2 GHz Dell workstation.

3 The Trouble With Browsers

The increase in number, size, and complexity of JavaScript applications underscores the need for a reliable execution infrastructure. In this section we show that modern browsers exhibit substantial reliability problems in the presence of demanding applications. In particular, we show that failures occur using both real and manufactured web pages, for three fundamental reasons: *(1)* poor failure isolation, when a bug in one application infects another, *(2)* scheduling starvation, where one overly aggressive application can prevent others from running, and *(3)* memory starvation, where the memory consumption of one application can prevent others from making good forward progress.

We demonstrate these reliability issues using several popular browsers, including Internet Explorer 6 (IE), Firefox 1.5, Opera 9, Safari 2, and Konqueror 3.5. Our browser tests were conducted in Windows XP SP2, Mac OS X 10.4, and Ubuntu Linux 6.0.6, for each platform on which a given browser was available. All tests were run on the same hardware, an Apple Mac Mini with a 1.66 GHz Intel Core Duo processor, configured to run each of the above operating systems.

3.1 Failure Isolation

Current web browsers do not prevent a crash in one browser-based application from crashing other browser-based applications. We find that such crashes usually cause the loss of the entire browser, including all open pages. Thus, as content on the web becomes more active and complex, it becomes more dangerous for users to load content from many different sources at the same time.

Browser-based application crashes are often caused by bugs in the browsers themselves, which are common. For example, a Bugzilla search for critical crash-related bugs in Firefox reported in 2005, narrowed to those the developers fixed, returns 25 bugs [7]. The same search for 2005 crash-related bugs in Konqueror returns 82 bugs [6]. As a specific example, in October 2006, we found that simply visiting `http://www.microsoft.com` crashed the Konqueror browser.[3] Such bugs illustrate the ease with which browser failures can occur.

These bugs in browsers indicate the need for a mechanism to prevent failures from leaking across applications.

[3]This corresponds to bug 90462 on `http://bugs.kde.org`, which has since been resolved.

3.2 Concurrency

Concurrency is a fundamental service for ensuring that applications remain responsive. It enables users to run several applications side by side, without allowing one application to starve another for CPU time. This is important for browser-based programs, as users frequently open many web sites at the same time. However, due to a lack of preemptive scheduling in many browsers, a long computation in one application can prevent any other application from making forward progress.

We have found many examples of CPU-intensive pages on the web. For example, some sites offer JavaScript and DHTML based animations, which can be sufficiently CPU bound to interfere with other open pages. One such page offers fireworks animations in JavaScript [24], which places enough demand on the CPU to cause a video from YouTube in another part of the browser to become choppy.

To quantify this, we measured page responsiveness by observing how long a JavaScript event handler was delayed on a test page while other pages were running. In tests of the JavaScript fireworks page above, launching one fireworks animation at a time increased Safari's average response time from zero to 300 milliseconds. Launching many of these animations can drive response times to over ten seconds, even on a dual core computer.

We can study concurrency problems in more detail with test pages in the lab. We built a test page that loads computationally intense scripts into three separate frames, each of which takes at least 500 ms to complete. We detect whether the browsers use preemptive scheduling by observing whether timestamps that are collected during each frame's computation are interleaved.

Figure 5 shows the results by graphing the collected timestamps from each frame. Surprisingly, in all observed browsers except Opera, JavaScript code is never interleaved across frames, indicating that most browsers do not offer preemptive scheduling. The same problem occurs across multiple open windows in Firefox, Safari, and Konqueror. Internet Explorer does interleave JavaScript execution across windows. These results show that web sites using significant JavaScript code may easily detract from the responsiveness of other web sites open in the browser.

Browsers attempt to deal with these concurrency problems in a variety of ways, but the results have mixed success. For example, all browsers offer a script engine timeout that opens a dialog box if a script attempts to use the CPU for longer than some threshold. The user can then choose to abort or continue the script. However, these thresholds are necessarily fairly high, allowing scripts to run for several seconds without interruption. Thus, sites can block user interaction with other web sites for seconds at a time. We have constructed denial of service pages that either briefly yield the CPU every few seconds or repeatedly launch computations in new frames. These pages leave other sites effectively unresponsive to user input.

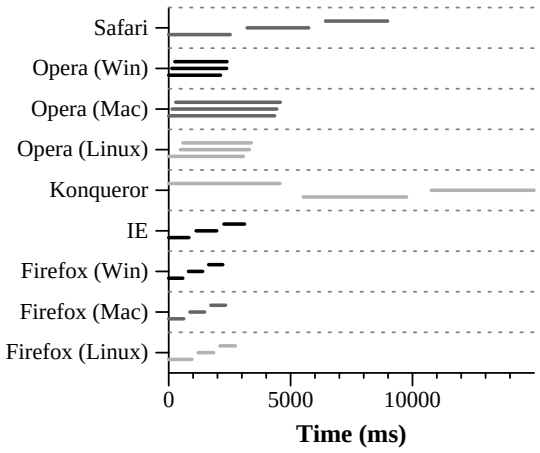

Figure 5: Concurrency behavior for popular web browsers. Each set of solid lines represents the execution of three frames loaded at the same time.

As another example, browsers attempt to improve page responsiveness using asynchronous network communication via the `XmlHttpRequest` class. However, this class also supports a synchronous mode in which the calling site blocks until a response is received from the server. In most browsers, a synchronous call blocks not only the page, but all interaction with the browser and any other open pages as well.

We verified this behavior using an adversarial page that makes a synchronous request to a server. The server then slept for one minute before returning a response. During this time period, all Firefox and Safari windows were left unusable, including all browser UI elements. In IE, the current window (including UI) was locked until the reply was received, while other windows remained responsive, and in Konqueror, all windows in an instance of the browser locked, including all UI elements. Only Opera was unaffected, where only the running code on the test page blocked.

Combined, these CPU bound pages and disruptive function calls prove that browsers need better support for scheduling to prevent starvation of other programs.

3.3 Memory Management

Memory management is a crucial service, as it allows multiple applications to share a scarce and necessary resource. Independently managing memory between applications is important to allow the allocation decisions of one application to remain isolated from others, and to allow users to terminate applications that grow too large.

However, we found that none of the web browsers we tested offered independent memory management for browser-based applications. Instead, one browser-based program can continuously allocate memory, leaving not just the offending program unresponsive, but the rest of the browser as well. In this situation, users cannot determine which browser-based program is responsible for the problem, nor can they terminate one browser-based program without losing others.

Memory leaks on real web sites have been common in practice. As noted by several web developers, JavaScript applications tend to be susceptible to memory leaks [25, 21]. Because interactive applications may stay open in the browser longer than traditional pages, these memory leaks may grow to enormous sizes. Tracking and correcting these leaks has proven challenging, because many JavaScript objects may not be garbage collected if they are registered with the DOM. Indeed, the documentation for the Google Maps API describes frequent memory leaks caused in earlier versions of the API [4].

We can demonstrate the adverse effects of these leaks using test pages in the lab. We use a string-doubling script described by Powell and Schneider [22], which quickly causes the browser to allocate enormous amounts of memory. In all browsers we tested, the page quickly allocated over 800 MB, often leaving the browser unresponsive and the rest of the system nearly unusable. Firefox refused to respond to input on all platforms and needed to be killed. Safari and Konqueror crashed. IE remained unresponsive and Opera remained slow until the offending windows were closed.

Because of common reports of memory leaks on the web and the dangers of causing failures across multiple applications, it is critical to introduce support for independent memory management in browsers.

3.4 Summary

In this section we have shown that existing web browsers are unreliable for executable content. As we have shown using both real world and lab examples, a failure (or even bad behavior) in one application can propagate through to others in several ways. In the next section, we describe a solution to the problem.

4 Reliability Through Isolation

In this section we show how processes executing in separate address spaces can solve the reliability problems described in the previous section. We present our architecture, which places content from different domains in different processes. We then explain how processes can safely isolate browser-based applications with low overhead, and we show how our architecture does not disrupt existing web sites. Finally, we describe how we have modified the Konqueror web browser to use our proposed architecture.

Our overall goal is to prevent browser-based applications from interacting with each other in undesirable ways. This raises two design issues. First, we must select an isolation mechanism to introduce into the browser that is effective and lightweight. Second, we must determine how to appropriately partition content using the mechanism.

We judge the success of our design by three criteria. The solution must be safe, ensuring the browser provides robust failure isolation, concurrency, and memory management for applications. It must be efficient, entailing reasonable time and space overhead in the browser. Finally, it must be backwards compatible, requiring no changes to existing web pages.

4.1 Architecture

At a high level, our solution is to modify existing web browsers to execute each browser-based application that a user visits in a separate OS process. This provides an effective and lightweight sandbox for each application.

However, it is difficult to precisely define a browser-based application. A single application may consist of multiple pages, frames, or HTTP connections. Conversely, a single page may contain multiple applications, loaded in different frames. We choose to define browser-based applications based on the *source* of the content. Specifically, we place all pages from a given second level domain (e.g., `yahoo.com` or `msn.com`) in the same process. Pages from different second level domains are then isolated from each other in different processes.

To accomplish this, we separate the browser's rendering engine from its user interface. For each domain that a user visits, we create a new rendering process, which draws HTML pages and runs JavaScript code. The browser's user interface (e.g., windows, tabs, menus) continues to operate in a single process that is separate from the rendering processes. This architecture is shown in Figure 6.

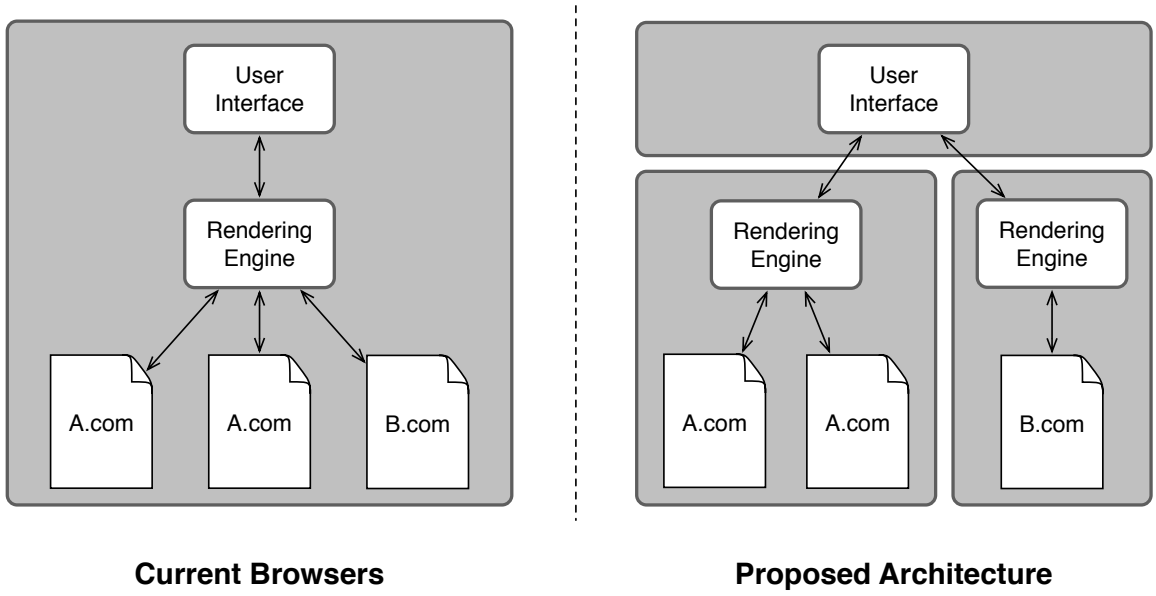

Figure 6: Current and proposed browser architectures. Gray boxes indicate process boundaries.

Our architecture is transparent to the user. That is, pages rendered in one process are visually mapped into the correct UI element of the user interface process. In this way, two different windows can show pages rendered by the same process, while a single window can show two frames rendered by different processes.

Our policies for creating and destroying processes are as follows. For each page a user requests, the browser looks at the domain of the page. If it has not already created a process for the domain, it launches a new rendering process. The browser then instructs the process to fetch and render the page, displaying the results in the current window or frame of the user interface.

As a user navigates links on a page or moves back and forward through the browser's history, the browser updates this mapping of processes to UI elements. Specifically, if a user navigates to a page from a different domain, the browser loads the requested page in its corresponding process and then maps the new page into the existing UI element. The previous page is suspended but kept alive in the background in case the user clicks the back button.

The browser must keep alive each rendering process and the pages it has drawn for as long as they exist in the browser's history. When a window is closed, the browser destroys all pages from the window's history. If all pages from a given domain are destroyed, the domain's process then exits. Note that as an optimization, the browser can set a threshold on the number of pages from the history to keep alive. If the user attempts to go back further than this threshold, the browser can simply reload any earlier pages from the web.

We can also augment browsers with existing process management tools. Using tools from the operating system, the browser can display a list of all domains, along with the current CPU utilization and memory usage for all content, broken down by domain. Users can then adjust the scheduling priority of the domain's process or define memory quotas, if desired. Most importantly, users can cleanly terminate the process of any misbehaving domain and continue to use the rest of the browser.

4.2 Discussion

We next discuss how our proposed architecture meets our three design criteria: safety, efficiency, and backwards compatibility.

In terms of safety, each process runs in a separate address space. Thus, crashes and other failures caused by one domain cannot adversely affect content from other domains. The operating system uses preemptive scheduling of processes, so expensive computations and blocked function calls on pages from one domain cannot starve pages from other domains. Also, processes support clean termination, allowing users to kill any process that becomes unresponsive due to a memory leak.

Processes are also efficient at the granularity we are using them. The time overhead for process startup and context switching is low compared to page rendering and JavaScript execution times. Similarly, the memory overhead of loading an HTML rendering engine in many processes can actually be quite low, due to shared libraries.

In order to be backwards compatible, it must be the case that applications that worked when running in a single address space browser continue to work in our new browser. The primary challenge is that pages may communicate with one another using shared memory. Therefore, we must ensure that those pages are placed in the same rendering address space.

To achieve this, we adopt a "same source" policy for assigning content to processes. This means that all pages from a given second level domain are rendered by the same process, while pages from different second level domains are rendered in different processes. For example, `http://www.yahoo.com` and `http://mail.yahoo.com` would be placed in the same process, but `http://mail.google.com` would be placed in a separate process.

This policy is intuitive, because the source of a web page is visible to the user. It is thus easy to reason about which pages will be isolated from each other. Additionally, our policy has the attractive property that a domain becomes responsible for the interaction between its pages, but these pages are safe from interference from pages from other domains.

Below, we show that the "same source" characterization is sufficient in practice to ensure that all pages that may communicate with each other via shared memory are placed in the same address space. Stronger policies would improve reliability at the expense of backwards compatibility, as they may place pages that could previously communicate in separate address spaces.

We now explain how our policy affects each of the known inter-page communication channels in web browsers.

- According to the "same-origin" security policy used in all browsers, two pages can communicate via shared memory (i.e., via the DOM) if they come from the exact same origin (i.e., same protocol, domain name, and port number) [23]. This channel is limited to "parent" and "child" pages, where a parent page opens a child page in a new window or an embedded frame. Current browser-based applications actively use this channel. We found 16 of 100 pages in our 2006 Alexa data set that had frames that could use this channel.

  Our policy uses a single address space for all pages from the same second level domain, which is a superset of pages from the same origin. Thus, we do not disrupt this communication channel.

- Parent and child pages can also communicate via shared memory if one page's origin is a suffix of the other's origin (e.g., `my.yahoo.com` and `yahoo.com`). This is only allowed if the page with the more specific origin grants permission, by modifying its `document.domain` variable to match the suffix. We found evidence of at least 9 pages in our 2006 Alexa data set using this channel.

  Because our policy applies to all content from a second level domain, pages can continue to communicate with other pages from a suffix of their own origin.

- Mozilla-based browsers support signed scripts, which allow a script to ask the user for certain permissions. Among these permissions is the ability to communicate with parent or child pages from any origin, via shared memory. However, we found no uses of this channel or of signed scripts in general in any of our data sets.

  Our policy does disrupt any signed scripts that attempt to communicate between pages from different origins. However, as this feature is not portable across browsers and does not appear to be used in practice, we argue that this disruption is an acceptable tradeoff for vastly improved reliability.

- The Opera browser supports a proposed HTML feature in which parent and child pages from any two origins can communicate via a message passing API [16]. We found no uses of this channel in any of our data sets.

  While our policy may place two pages using this mechanism in separate address spaces, it is still possible to support this communication channel. This is because message passing is a restricted form of communication which is natural to support with inter-process communication.

- Any pages from the same origin can communicate via cookie values. This channel depends on the filesystem rather than shared memory, so it is unaffected by process boundaries or our policy.

- Unintentional side channels have been shown to exist between pages, using the browser cache and visited link history to encode information [19]. Such channels are also unaffected by process boundaries or our policy.

For these reasons, our "same source" policy maintains backwards compatibility in practice, as we know of no sites which it disrupts.

4.3 Implementation

We have implemented our system by modifying the Konqueror web browser for KDE, running on Fedora Core 5 Linux. The Konqueror browser is reasonably well designed, so we were able to implement our "same source" policy in less than one thousand lines of code.

Our choice of Konqueror was primarily out of convenience, but we are currently investigating how to modify other popular browsers as well. For example, we have proposed this architecture to developers of Firefox 3 [3]. One challenge for Firefox is that the browser's profile data, including bookmarks, caches, cookies, and preferences, are designed to be single threaded. Allowing multiple processes to concurrently access the profile could cause data corruption [1]. We are hopeful that the interface to the profile data can be made thread-safe, so that Firefox can benefit from this architecture as well.

5 Evaluation

In this section we evaluate our design in terms of its benefits and costs, including safety, backwards compatibility, and efficiency. Using our prototype implementation, we show that our new browser *(1)* does not crash on pages that crashed the original browser, *(2)* does not disrupt the top 100 pages from the Alexa data set (beyond a few minor bugs in our implementation), and *(3)* imposes minimal latency and space overhead.

5.1 Safety

We first evaluate whether our implementation is successful at defending against the reliability problems we have identified. To do this, we loaded the pages described in Section 3 in our prototype browser. Our results are summarized in Table 1.

The table shows how both current browsers and our prototype browser behave on the pages we tested. Checkmarks indicate when these pages did not interfere with pages from other domains. We acknowledge that we have introduced the distinction between domains as a reliability metric, perhaps creating an unfair comparison. However, we also argue that this distinction is both justified and desirable, from our discussions in Section 4. That is, it provides maximal reliability while maintaining backwards compatibility.

For failure isolation, we tested the Konqueror bug mentioned in Section 3. Visiting `http://www.microsoft.com` in the original Konqueror browser caused the entire browser to crash. Visiting it in our prototype only crashed the process rendering the page, which simply left a blank tab in the browser.

For concurrency, we measured the response time for event handlers when either our CPU adversary page or the JavaScript Fireworks demonstration page were loaded in tabs in the browser. In Konqueror, the average response times in the presence of these pages were 840 ms and 170 ms, respectively. In our prototype, the response times were 4.1 ms and 3.6 ms, respectively. This indicates substantially higher responsiveness in the face of heavy computation. Qualitatively, we found that we could easily interact with web pages in other tabs even in the face of sites with heavy computation, unlike in Konqueror.

We also tested our synchronous network request adversary page. This page fully locked the UI for Konqueror, but it only locked the contents of the offending page in our prototype. The prototype's user interface and all other pages remained responsive.

For memory management, we tested our memory leak test page. As the page allocated all of the system's memory, it caused the rest of the system to become slow. However, while the original Konqueror locked and eventually crashed, our prototype browser remained responsive. Terminating the process of the offending page was sufficient reclaim all memory it had allocated.

5.2 Backwards Compatibility

To empirically test the backwards compatibility of our prototype, we loaded the top 100 pages from the 2006 Alexa data set. We compared them visually with the same pages in an unmodified instance of Konqueror to confirm that page rendering was not disrupted.

The original Konqueror browser was unable to render every page properly, as some pages have browser-specific features. For this reason, there were 12 pages which caused JavaScript errors in both the original and prototype browsers. Also, `http://www.microsoft.com` was among the sites we visited, causing the browser to crash. Our prototype browser con-

	IE	Firefox	Safari	Opera	Konqueror	Prototype
Microsoft.com	✓	✓	✓	✓	✗	✓
CPU Adversary Test	✗	✗	✗	✓	✗	✓
JavaScript Fireworks	✗	✗	✗	✓	✗	✓
Synchronous XHR Test	✗	✗	✗	✓	✗	✓
Memory Leak Test	✗	✗	✗	✗	✗	✓

Table 1: Demonstrated interference across domains. Crosses indicate pages that interfered with content from other domains in the browser. Checkmarks indicate when the behavior of these pages was isolated from other content.

fined these crashes and errors to a single tab of the browser.

Due to a bug in our implementation, our current prototype was unable to handle pages that communicated across frames. This bug does not represent an architectural problem, and we expect to resolve it shortly. As a result of the bug, we encountered 6 pages that caused JavaScript errors in our prototype which worked in the original browser. Also, we found that some embedded frames were drawn with unnecessary scroll bars or outside the browser window. These issues can also be resolved with slight improvements to our implementation, and do not represent architectural flaws.

5.3 Efficiency

We measured the time and memory overhead of our prototype relative to unmodified Konqueror. First, we measured the time to start the browser and load a small test page. We found a 41% increase in startup time, from 1.51 seconds to 2.12 seconds. We then followed a link to a small page from a different domain, which instantiated a new process in our prototype browser. Here, page load time increased from 124 milliseconds to 890 milliseconds. We believe we can remove the majority of this time with simple caching of pre-forked processes, but we have not yet implemented this feature in our prototype. However, in the context of common page load times, the additional 0.9 seconds is relatively small.

Second, we measured the memory overhead introduced for each new rendering process in our prototype. We compared our prototype's behavior to running a single instance of Konqueror for all pages and to running a separate instance of Konqueror for each page. We used the Linux `pmap` tool to observe writeable/private memory for all processes used by a given browser.

Table 2 displays the results; values shown for loading an additional page are the averages across four page loads. Our prototype uses 24% more memory for a single blank page than the original Konqueror, as the user interface runs in a separate process than the page. For each additional blank page opened, however, the prototype is 46% more efficient than starting a new Konqueror process from scratch.

For all alternatives, we note that the overhead is sufficiently low to believe users can open a large number of sites without difficulty on modern computers. This suggests a low cost for adding substantial reliability.

6 Related Work

There has been some effort to support web browsing with multiple processes in current browsers, but existing approaches do not use processes at an appropriate level of granularity. We discuss these approaches along with other related work, including process-based isolation in other systems, robust runtime environments for browser-based applications, and designs for more robust and secure web browsers.

6.1 Processes in Current Browsers

Some current web browsers allow users to instantiate multiple browser processes, but they do so at an inappropriate granularity. Internet Explorer and Konqueror both allow groups of windows to belong to different processes. Specifically, each time the user starts the browser from its icon on the desktop, a new browser process is started. Opening a new window or tab from within the browser loads another page in the same process.

By manually starting several browser processes, users can mitigate the effects of failures in other pages. A browser crash only affects the pages in the same process, and the operating system handles concurrency between pages from different processes. Memory leaks can also be contained, as users can cleanly terminate a process containing a page with a leak.

Page	Konq. (Same Process)	Prototype	Konq. (Diff Processes)
Single blank page	5.6	7.0	5.6
Additional blank page	0.29	3.0	5.6
Additional Google Maps	4.3	9.6	11.5

Table 2: Private memory allocated per page, in megabytes

However, current browsers are partitioned into processes according to user interface elements and not content. Thus, they do not isolate browser-based applications, but rather groups of unrelated applications. Pages from different applications may often exist in the same process, forcing them to share a common fate. Additionally, there is no visual indication of which pages belong to which process, leaving users unable to make informed decisions about the impact any failure will have. This is particularly problematic for deciding which browser process to terminate, if one page misbehaves.

On a similar note, Firefox can launch one process per user profile. This may allow users to start multiple isolated instances of the browser, but it comes at a great cost for functionality: each profile maintains its own bookmarks, preferences, and cache data, which cannot be shared across profiles. Thus, it is awkward to use multiple processes within Firefox. All other browsers we have tested are limited to a single process model.

6.2 Process-Based Isolation

Many researchers have investigated how to use processes to improve the isolation in other runtime environments. For example, KaffeOS incorporated process-based isolation into the Java Virtual Machine, while allowing direct sharing of objects [10]. A number of other efforts have added a processes or other protection domains to the JVM as well [26, 15].

Researchers have also proposed using Java *isolates*, which are Java application components that do not share state [20]. Isolates offer many of the benefits of process-based isolation, allowing multiple independent applications to run reliably in the same JVM. This mechanism has been used, for example, to support a resource management API for Java applications [13].

Additionally, some have asserted that the lack of processes in the original JVM led to a number of challenges in the design and implementation of the system (J. Waldo, personal communication, November 7, 2006).

These efforts are complimentary to our work, showing the need for effective isolation in runtime environments in general.

6.3 Browser Runtime Environments

In the context of web browsers, there has been notable work in providing robust client environments for running applications. Over a decade ago, Sun introduced Java applets [14], which provided all the benefits of a Java Virtual Machine to code running in the web browser: concurrent threads, type safety, relatively high performance, and security mechanisms. More recently, Microsoft has moved to support .NET applications inside web browsers, using the Windows Presentation Foundation (WPF) [11].

However, while applets provided a more robust runtime environment for application code than JavaScript, they have not become ubiquitous. In fact, in sharp contrast to the recent increases in JavaScript usage, not a single page in the top 100 lists contains a Java applet, from 2003 to 2006. We discovered four Java applets in the Alexa top 500 global domains: two were "hidden" applets for tracking web client behavior, and two were chat applications. This scarcity of applets indicates a strong preference among web developers for using JavaScript rather than applets to build active web content.

Our work differs from efforts to introduce new runtime environments into the browser, as we focus instead on making the browser's own runtime environment more reliable. This is important, given the large number of JavaScript-based applications on the web.

In other work to improve the JavaScript runtime environment, Adobe has recently donated its ActionScript virtual machine and just-in-time compiler to Mozilla [9]. (Both JavaScript and ActionScript are implementations of the ECMAScript standard [17].) Dubbed Tamarin, this virtual machine could allow higher performance for future versions of the JavaScript language. However, without the isolation provided by separate address spaces, it is unlikely that this will prevent many of the reliability problems we have discussed.

6.4 Robust and Secure Browsers

Finally, researchers have attempted to improve browser robustness and security through other changes to browser architectures. Cox et al designed Tahoma [12], which isolates each web application in its own virtual machine, protecting both other web applications and the operating system from its behavior. We choose to isolate applications in the browser using OS processes instead, offering a lower cost alternative for reliability. Our approach does require greater trust in the browser's own security, though. Similarly, Ioannidis and Bellovin propose a secure web browser using SubOS processes [18], although they offer little insight into the appropriate granularity or interaction between processes. In contrast, we argue for a choice of process boundaries which offers high reliability without breaking existing sites.

7 Conclusion

Through measurements of both web content and browser behavior, we have shown that current web browsers provide an unreliable environment for running an increasingly popular class of applications. Browser-based applications are rapidly spreading, yet browsers do not isolate them from each other. This leads to critical problems with respect to failure isolation, concurrency, and memory management.

We have shown that browser-based applications can be safely isolated from each other using OS processes. Processes prevent unwanted interactions between programs in the browser, and they are efficient relative to other browser operations, both in time and memory overhead. By using a "same source" policy for assigning pages to processes, we can also preserve backwards compatibility with existing web sites.

Our prototype browser has demonstrated these properties. It continues to run after loading pages that cause other browsers to fail, and our current implementation does not disrupt most pages we have tested. As a result, we encourage developers of other web browsers to adopt our proposed architecture to provide reliable environments for active web content.

References

[1] Bug 135137 - profile data cannot be shared by muliple running instances. `https://bugzilla.mozilla.org/show_bug.cgi?id=135137`, Apr. 2002.

[2] Alexa web search - top 500. `http://www.alexa.com/site/ds/top_500`, 2006.

[3] Firefox/feature brainstorming. `http://wiki.mozilla.org/Firefox/Feature_Brainstorming`, Oct. 2006.

[4] Google maps api documentation. `http://www.google.com/apis/maps/documentation/`, 2006.

[5] Internet archive: Wayback machine. `http://web.archive.org`, 2006.

[6] Kde bug tracking system. `http://bugs.kde.org/query.cgi`, 2006.

[7] mozilla.org bugzilla. `https://bugzilla.mozilla.org/query.cgi`, 2006.

[8] Netcraft - most visited web sites. `http://toolbar.netcraft.com/stats/topsites`, 2006.

[9] Tamarin project. `http://www.mozilla.org/projects/tamarin/`, Nov. 2006.

[10] G. Back, W. C. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management, and sharing in java. In *OSDI*, Oct. 2000.

[11] K. Corby. Windows presentation foundation on the web: Web browser applications. `http://msdn.microsoft.com/library/en-us/dnlong/html/wpfandwbas.asp`, Oct. 2005.

[12] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[13] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A resource management interface for the java platform. Technical Report TR-2003-124, Sun Microsystems, May 2003.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, June 1998.

[16] I. Hickson. Web applications 1.0: Cross-document messaging. `http://whatwg.org/specs/web-apps/current-work/#crossDocumentMessages`, 2006.

[17] E. International. Standard ecma-202: Ecmascript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, Dec. 1999.

[18] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, June 2001.

[19] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International Conference on World Wide Web (WWW 2006)*, May 2006.

[20] Java Community Process. Jsr 121: Application isolation api. http://jcp.org/en/jsr/detail?id=121, June 2006.

[21] P.-P. Koch. Quirksblog - memory leaks. http://www.quirksmode.org/blog/archives/coding_techniques/memory_leaks/index.html, Sept. 2006.

[22] T. Powell and F. Schneider. *JavaScript: The Complete Reference*. McGraw-Hill/Osborne, 2004.

[23] J. Ruderman. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2001.

[24] S. Schiller. Fireworks.js: A dhtml fireworks effect. http://www.schillmania.com/projects/fireworks/, 2006.

[25] R. Stratulat. Biggest ajax problem. http://www.stratulat.com/blogs/index.php?title=biggest_ajax_problem&more=1&c=1&tb=1&pb=1, July 2006.

[26] P. Tullmann and J. Lepreau. Nested java processes: Os structure for mobile code. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sept. 1998.