ScriptSpaces: An Isolation Abstraction for Web Browsers

Amarjyoti Deka

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Applications

Godmar V. Back, Chair
Barbara G. Ryder
Eli Tilevich

July 22, 2010
Blacksburg, Virginia

Keywords: Web Browsers, Isolation, JavaScript, Resource Control
Copyright ©Amarjyoti Deka 2010
ScriptSpaces: An Isolation Abstraction for Web Browsers
Amarjyoti Deka

ABSTRACT

Current web browsers are ill-prepared to manage execution of scripts embedded in web pages, because they treat all JavaScript code executing in a page as one unit. All code shares the same namespace, same security domain, and shares uncontrolled access to the same heap; some browsers even use the same thread for multiple tabs or windows. This lack of isolation frequently causes problems that range from loss of functionality to security compromises.

ScriptSpace is an abstraction that provides separate, isolated execution environments for parts or all of a web page. Within each ScriptSpace, we maintain the traditional, single-threaded JavaScript environment to provide compatibility with existing code written under this assumption. Multiple ScriptSpaces within a page are isolated with respect to namespace, CPU, and memory consumption. The user has the ability to safely terminate failing scripts without affecting the functionality of still-functional components of the page, or of other pages.

We implemented a prototype of ScriptSpace based on the Firefox 3.0 browser. Rather than mapping ScriptSpaces to OS-level threads, we exploit a migrating-thread model in which threads enter and leave the ScriptSpaces associated with the respective sections of the document tree during the event dispatching process. A proportional share scheduler ensures that the number of bytecode instructions executed within each ScriptSpace is controlled. Our prototype can isolate resource-hogging gadgets within an iGoogle Mashup page as well as across multiple pages loaded in the browser and still retain interactive response.
Acknowledgments

I wish to thank Dr. Godmar Back for being my advisor for this project for the last two years. Without his constant mentoring and encouragement the project would have never reached the stage as it is today. I feel honored to be his student.

One of the primary reasons for joining Virginia Tech for my higher studies was to be part of cutting edge research and development activities in Computer Science. I have thoroughly enjoyed that part taking different courses, participating in various departmental seminars and at different times interacting directly with the faculty members of the department. I am indebted to my department for all of these wonderful things.

I wish to thank Dr. Barbara Ryder and Dr. Eli Tilevich for agreeing to be on my committee and for their feedback during regular research group meetings, and casual conversation.

I have always enjoyed the cohesiveness and enthusiasm shown by my fellow students. I wish to thank my fellow graduate students Ajeet Singh, Hari Pyla, Yang Jiao and Yannick Verdie for their fun filled company.

Life would have been difficult without Rubee, my wife, who has always encouraged me to work hard. I wish to thank her for her constant support. Lastly, I will take this opportunity to thank my parents without whose support I would have never come to US for my studies.
## Contents

- **List of Figures** vi
- **List of Tables** vii

### 1 Introduction
- 1.1 Outline ........................................ 5

### 2 Background
- 2.1 JavaScript Execution Model in Web Browsers .......................... 6
- 2.2 Web 2.0 Mashups .................................. 13
- 2.3 Principles of Isolation and Resource Management ..................... 14
  - 2.3.1 Isolation in Web Browsers ................................ 17
- 2.4 CPU Scheduling .................................... 18
- 2.5 Models for Concurrency in Software Systems ......................... 22
  - 2.5.1 Event-Based Concurrency ............................. 22
  - 2.5.2 Thread-Based Concurrency ............................. 23
  - 2.5.3 Hybrid Approaches for Concurrency ..................... 23
  - 2.5.4 Migrating Thread Model ............................. 23

### 3 Design
- 3.1 Isolation using ScriptSpaces .................................. 25
  - 3.1.1 ScriptSpace Association .............................. 25
- 3.2 User/Kernel Boundary in the Browser ............................. 27
List of Figures

2.1 Example of a Document Object Model (DOM) tree ............................ 8
2.2 Example of an event flow for a click event .................................... 10
2.3 BVTS example ............................................................................. 21

3.1 Example of ScriptSpace association ................................................. 27
3.2 User/Kernel boundary in web browsers .......................................... 28
3.3 Event-based concurrency ................................................................. 30
3.4 Migrating thread for concurrency .................................................... 31

4.1 Overview of changes made to Firefox .............................................. 35
4.2 Main event loop in Firefox .............................................................. 37
4.3 Processing of script invoking events in modified Firefox ................. 38
4.4 Flow chart for scheduler entry ......................................................... 43
4.5 Flow chart for scheduler exit .......................................................... 44
4.6 SSManager v1.0 UI ...................................................................... 46

5.1 Progress of scripts associated with a single ScriptSpace ................. 49
5.2 Progress of scripts associated with separate ScriptSpaces ................ 50
5.3 Illustration of CPU resource control .............................................. 52
5.4 Service error graph .................................................................... 53
5.5 Graph for DOS attacks in vanilla Firefox ...................................... 55
5.6 Graph for DOS attacks in modified Firefox .................................... 56
5.7 Execution of scripts in iGoogle gadgets in vanilla Firefox .............. 58
5.8 Execution of scripts in iGoogle gadgets in modified Firefox ............ 59
List of Tables

1.1 Behavior of browsers under CPU and memory load ..................... 3

2.1 JavaScript APIs provided by mashups .................................. 14

4.1 nsIScriptSpace API .................................................. 39

4.2 nsIScriptSpaceRuntime API .......................................... 41
Chapter 1

Introduction

Browsers are applications for viewing and interacting with web content. The majority of web content today is a combination of static HTML [W3C99] and scripts. HTML is used to describe the text, multimedia and hyperlinks present in a web page for the browser. Scripts, which are programs written in JavaScript, are used to improve usability of the static HTML. More than 70 out of the 100 most popular websites reported by Alexa [Ale10] use 10KB or more JavaScript code in their home pages [RBGL07]. Scripts are executed in response to user interactions with the web page such as clicking a button, pressing a key, etc. JavaScript is used to write programs to perform tasks ranging from user input validation to complex animation.

A JavaScript Virtual Machine interprets and executes code written in JavaScript. A browser integrates a JavaScript virtual machine to execute scripts embedded in web pages. As a platform for the execution of scripts, the browser assumes the role of an operating system. It provides resources such as CPU and memory to the executing scripts. Such software systems that act as a platform for the execution of user code require a robust isolation mechanism that carefully protects the execution boundaries of each unit of code and the system itself so that their execution can be independently controlled and terminated. An isolation mechanism also facilitates the management of the resources consumed by each isolated unit. Operating systems use robust abstractions such as processes to isolate executing programs from each other, facilities which most existing browsers lack. Therefore a malicious script embedded in a web page can exhaust the CPU or memory allocated for the browser and prevent the browser from loading or processing other content.

To explore how current browsers react in the presence of resource denial attacks, we ran a
set of CPU and memory bound JavaScript codes in a set of popular web browsers. The code snippet shown in 1.1 is the simplest possible CPU hogging script, using an infinite loop. Similarly, the code snippet in 1.2 creates a memory hog situation by allocating strings whose size doubles with each iteration of the inner loop.

```
<script type="text/javascript">
function cpubound()
{
    while(true);
}
// register the script 'cpubound' as load event handler of the
document. The event will be triggered as soon as the document
// is completely loaded in the browser.
document.addEventListener("load", cpubound, false);
</script>
```

Listing 1.1: Example of a simple CPU bound script in JavaScript

```
<script type="text/javascript">
for (var i = 0;; i++){
    // register one shot timers to the current window object
    // where the HTML document is being loaded.
    window.setTimeout(function () {
        // The function will allocate memory exponentially
        // by allocating a linked list of
        // nodes containing strings
        // X, XX, XXXX, in the attribute 'value'
        // and the reference to the
        // next node in the attribute 'next'.
        var o = { value: "X" };
        for (;;){
            o = { next: o, value: o.value + o.value };
        }
    }, i++);
}
</script>
```

Listing 1.2: Example of a memory hogging script in JavaScript

We added each of the scripts to two different web pages and recorded the browser behavior while the web pages loaded. We observed whether we were able to use the system area of the browser while executing the loads. The system area is the portion of the browser that contains the browser menus and the back, forward, stop, and refresh buttons of the browser. We also observed whether we could right-click to display the context menu in the client area, which is the portion of the browser that displays the loaded web page. We recorded 

*Responsive* if we could do so or *Unresponsive* otherwise. Some browsers are able to detect
CPU load generating scripts and display a prompt asking user permission to stop the script execution. We recorded *Detected* if the browser displayed such a prompt and *Undetected* otherwise. Table 1.1 shows the results we obtained.

Table 1.1: Behavior of browsers under CPU and memory load

<table>
<thead>
<tr>
<th>Browser</th>
<th>System Area</th>
<th>Client Area</th>
<th>CPU Load</th>
<th>Memory Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 3.0.1</td>
<td>Unresponsive</td>
<td>Unresponsive</td>
<td>Detected</td>
<td>Undetected</td>
</tr>
<tr>
<td>IE 7</td>
<td>Unresponsive</td>
<td>Unresponsive</td>
<td>Detected</td>
<td>Undetected</td>
</tr>
<tr>
<td>Safari 3.1.2</td>
<td>Unresponsive</td>
<td>Unresponsive</td>
<td>Detected</td>
<td>Undetected</td>
</tr>
<tr>
<td>Opera 9.51</td>
<td>Responsive</td>
<td>Responsive</td>
<td>Undetected</td>
<td>Undetected</td>
</tr>
<tr>
<td>Chrome 02.149.29</td>
<td>Responsive</td>
<td>Unresponsive</td>
<td>Undetected</td>
<td>Undetected</td>
</tr>
<tr>
<td>IE8</td>
<td>Responsive</td>
<td>Unresponsive</td>
<td>Detected</td>
<td>Undetected</td>
</tr>
<tr>
<td>Firefox 3.5</td>
<td>Unresponsive</td>
<td>Unresponsive</td>
<td>Detected</td>
<td>Undetected</td>
</tr>
</tbody>
</table>

We found that the system areas of Firefox, Safari and IE7 became unresponsive in both cases. For example, we could not create a new tab, which is the UI element in the browser that displays a web page, as the create-tab button was not responding to mouse clicks.

In order to allow the browser to be responsive even if individual tabs fail, some recent browsers have started using a multi-process architecture. For example, Google Chromium [Moz09] and IE8 [Zei08] use OS processes to isolate pages in tabs. These browsers load the system area of the browser and the tabs in separate processes so that any failure in one of the tabs can be isolated from the rest of the browser. The system area of these browsers remained responsive in our experiment. However, a process-per-tab approach is not sufficient for isolating components within a web page. Web 2.0 Mashups such as igoogle.com, facebook.com integrate code and data from multiple sources to provide different features within the same page. If any of the features includes malicious or buggy code that makes unbounded use of resources, the process-per-tab strategy will still result in the entire mashup page being unusable, instead of detecting the failed feature and disabling it.

Table 1.1 indicates that none of the browsers in the experiment is able to detect the memory exhausting scenario. Some browsers are able to detect excessive CPU iterations, but they do not have a control mechanism to mitigate the impact beyond simply stopping the script.

These challenges motivated us to study the problem of isolation in a web browser environment. This thesis presents ScriptSpaces, an abstraction for the isolation of script execution
in web browsers that isolates script executions across multiple web pages as well as within a single page.

ScriptSpaces are created for browser tabs that separate content in different web pages and for HTML containers like *div*, *frame* and *iframe* that separate different content within a web page. In this model, every script embedded in a web page is associated with a ScriptSpace so that the execution of one script can be separated from another based on its associated ScriptSpace.

JavaScript as a language does not have any concurrency provisions. Therefore, within a ScriptSpace, we maintain the traditional, single-threaded JavaScript environment to provide compatibility with existing code written under this assumption. However, multiple ScriptSpaces within a page are isolated with respect to namespace, CPU, and memory consumption. Therefore, multiple CPU-bound activities in different ScriptSpaces can execute simultaneously, preventing a long running or infinitely looping script from “locking up” the page.

A ScriptSpace acts as the resource principal which provides the CPU and memory resources during the execution of an associated script. Because the execution of scripts in the browser is event driven, a fixed association of a process or an OS level thread to a ScriptSpace is not suitable. Instead, we use a migrating thread model [FL94] where a thread enters into and exits a ScriptSpace as it executes the scripts associated with that ScriptSpace. Resources consumed during the execution of a script are charged to the associated ScriptSpace. Any resource allocation request by an executing script can then be granted or denied by evaluating the request against the resource usage and the usage limits associated with the ScriptSpace.

ScriptSpaces provide a user the ability to safely terminate failing scripts without affecting the functionality of still-functional components of the page, or of other pages. Termination requests for script execution are handled from safe-points in the user script code to ensure that termination does not leave the browser in an inconsistent state. A ScriptSpace can be disabled at any point to prevent the execution of any script associated with the ScriptSpace. Failure isolation at the level of a ScriptSpace ensures that the remaining ScriptSpaces in the page are still be able to provide resources to the scripts associated with them. The rest of the web page, therefore, remains responsive to user interaction.

We implemented a prototype based on the Firefox 3.0 browser to assess the feasibility of our design. In our prototype, a proportional share scheduler [NVZ01] ensures that the number
of bytecode instructions executed within each script space is controlled. We also developed a CPU management extension for Firefox based on the resource control API exposed by our implementation. The extension provides a UI to monitor and control the CPU consumption of executing scripts in real time.

We observed the behavior of the resultant browser under CPU loads generated by scripts within a page and in multiple pages opened in different tabs in the browser. We recorded the amount of work done by individual scripts to determine whether they were able to receive CPU time proportional to the CPU share assignments of their associated ScriptSpaces. In order to show that a CPU bound script execution did not starve other scripts of CPU time, we loaded web pages with periodically running scripts while the browser was executing CPU bound scripts and verified the absence of starvation for the periodically running scripts.

We observed that our implementation was able to allocate CPU time to the CPU bound scripts proportional to the assigned share of their associated ScriptSpaces with a fairly low error. In our experiments, the error was less than 6% of the total work done per time unit. We verified the ability to change the CPU allocation for a script by varying the CPU share for the associated ScriptSpace.

We confirmed that our prototype can thwart denial-of-service attacks aimed at CPU consumption, unlike the existing Firefox browser. We verified our ScriptSpace enhanced version of Firefox can isolate resource-hogging gadgets within an iGoogle Mashup page in the browser and still retain interactive responsiveness.

1.1 Outline

In Chapter 2, we discuss the key technologies and concepts to which the following chapters refer. We discuss the design principles behind the ScriptSpace abstraction in Chapter 3. In Chapter 4, we discuss the implementation of our prototype using Firefox as our testbed. The discussion also covers the ScriptSpace API that we expose to the user and the SSManager extension that is built using the API. We evaluate our implementation in Chapter 5. Chapter 6 provides a discussion of related work. Chapter 7 suggests directions for future work and summarizes our contributions.
Chapter 2

Background

Our isolation abstraction is designed for isolating script execution in web browsers. We discuss the model of execution of scripts in web browsers in detail in section 2.1. Web 2.0 mashup websites provide a good use case for the isolation provided by ScriptSpaces. We discuss Web 2.0 mashups in section 2.2. Execution isolation is also a common issue encountered in other software systems. We discuss approaches used in operating systems, language runtime systems, and existing web browsers in section 2.3. We discuss proportional share CPU scheduling in section 2.4 as we have used one such scheduler in our implementation. Finally, we provide a general discussion of the concurrency models in server systems in section 2.5 as we have based our implementation of user event processing on those models.

2.1 JavaScript Execution Model in Web Browsers

A web page consists of data such as text and images, as well as script code specified within elements defined by the HTML specification [W3C99]. The HTML specification defines the script element for including JavaScript code within a web page. An HTML parser in the browser parses the web page and passes the scripts to the JavaScript engine embedded in the browser for processing. Depending on how these scripts are executed, the scripts can be classified into two types:

- **In-line Scripts**, loaded and executed during the loading of the web page.
- **Event Handlers**, which are scripts triggered by user interaction with the web page once it is loaded.
Inline Scripts

There are two ways inline scripts can be embedded in a web page. First, a developer can write JavaScript code within a `script` element. For example, the web page 2.1 includes a `script` element containing a script that prints the current date in the web page. Second, the developer can specify a location from where the JavaScript code should be downloaded within the optional `src` attribute of the `script` element. For example, JavaScript libraries that support animations can be included in a web page by specifying their URL in the `src` attribute. In both of these cases, the JavaScript runtime interprets the parsed JavaScript content. For the example in 2.1, the JavaScript interpreter initializes the value of the variable `now` with the current date and invokes the `write` method of the `document` object. The execution of inline scripts happens only once during the lifetime of the page.

```html
<html>
<body>
<h1>
<script type="text/javascript">
var now = new Date();
// populate the content of the document with
today's date.
document.write("Today is:" + now.getMonth() + "/" + now.getDate() + "/" + now.getFullYear());
</script>
</h1>
</body>
</html>
```

Listing 2.1: Example Inline Script Execution

Event Handlers

The Document Object Model or DOM defines a convention for representing and interacting with a document written in HTML and XML such as a web page [W3C09a]. The DOM represents a document as a tree-based organization of the HTML content in the document. For example, Firefox builds the DOM tree shown in figure 2.1 for the following simple HTML page in 2.2.

```html
<html>
<head>
<title>Simple Page</title>
</head>
```
Each node of the DOM tree is an object with a set of attributes and methods. Browsers provide JavaScript language bindings for DOM interfaces which can be used to access and manipulate the DOM nodes from scripts in the web pages.
DOM Events are a set of events that are generated by user interaction with the web pages [W3C09b]. *Click, keypress, load* are examples of DOM events which are generated when the user clicks a mouse button, presses a key, or to signal that the content of the page is completely loaded. An event is associated with a target node with which the user currently interacts. For example, for a key press event, the target node is the node that currently has the focus.

DOM nodes in a browser implement the `EventTarget` DOM interface and provide a JavaScript binding for it. The `addEventListener()` and `addEventListenerNS()` methods of the interface can be used to register an event handler to a node for an event identified by the `type` parameter passed to these methods. For example, the code snippet 2.3 will register the function `clickHandler` on the button identified by the id `test`.

An event handler can also be registered as a property of a DOM node. For example, the same event handler can be registered using the `onclick` property of the button as shown in the code snippet 2.4.

```
// event handler for click event
function clickHandler()
{
    alert("click handler invoked");
}
// register the event handler with
// DOM element identified by id "test"
document.getElementById("test").addEventListener("click",
    clickHandler, false);
```

Listing 2.3: Registering an event handler to a button using `addEventListener`

```
// event handler for click event
function clickHandler()
{
    alert("click handler invoked");
}
// register the event handler ‘‘clickHandler’’ as the onclick
// property of the input button node.
<input id="test" type="button" value="click me"
    onclick="clickHandler();" />
```

Listing 2.4: Registering an event handler to a button using its `onclick` property

The handling of an event in the web browser can cause multiple handlers to execute. Once an event is generated by user interaction, the event *flows* through the DOM sub-tree containing
the ancestor nodes of the target node in the original tree, executing event handlers at each node. Different flows are known as *phases* of the event processing. During the *CAPTURING PHASE* of the event flow, the event flows from the root node to the target node and the event handlers registered for the capturing phase of the event are executed. Then the handlers associated with the target node are executed. This is known as the *AT_TARGET phase*. The event finally bubbles up the sub-tree in the *BUBBLING PHASE*, executing any event handler registered for the bubbling phase at each node. As an example, a click on the button in the code snippet 2.2 will result in the event flow shown in figure 2.2.

![Figure 2.2: Example of an event flow for a click event](image)

The *DOM Event* interface provides an API to manipulate the flow. For example, an event flow can be interrupted by calling the *stopPropagation()* method on the event object. Browsers may provide a default handler of an event, which is executed only after the event flow completes. For example, the browser can provide a default implementation for the click
event that navigates the user to a new page after the user clicks on a link. Such default behavior can be prevented using the `preventDefault()` method of the `Event` interface.

Events can be nested. The handling of one event may generate one or more events. For example, a click handler associated with a button can change the color on the button. This change will trigger a `DOMAttrModified` event for the button in the Firefox browser.

There are two other types of events besides DOM events. The first type is used for the network events generated by the `XMLHttpRequest` interface [W3C08]. The `XMLHttpRequest` or XHR interface is used in many websites to develop dynamic and responsive content using AJAX [Gar05]. XHR enables a script to send HTTP requests to the server associated with the web page and receive responses asynchronously via a response handler. It is used to implement AJAX as it allows partial updates of the web page with the content retrieved asynchronously in the response handler. Internally, the responses for XHR generate `readystatechange` events to the request object. As a result, the `onreadystatechange` handler of the request object is executed. For example, the following web page retrieves the content in `sample.txt` located in the server hosting the web page using the `XMLHttpRequest` interface, and then displays it:

```html
<script type="text/javascript" language="javascript">
function buttonpressed()
{
    // create an XHR object for sending asynchronous HTTP request
    var xhr = new XMLHttpRequest();
    xhr.overrideMimeType('text/xml');
    // register the anonymous function as the response handler to receive the HTTP responses.
    xhr.onreadystatechange = function()
    {
        if(xhr.readyState == 4)
        {
            if(xhr.status == 200) {
                // display the data received as HTML as well as in a dialog
                document.getElementById("A").innerHTML = xhr.responseText;
                alert("Received:" + xhr.responseText);
            }
            else
            
                alert("Error code " + xhr.status);
        }
    }
    // create a GET HTTP request to retrieve
</script>
```
Listing 2.5: Example JavaScript Execution via XMLHttpRequest

Most modern browsers also support timer events. The `setTimeout()` and `setInterval()` functions may be used to execute any JavaScript function after a timeout. The `setTimeout` API is used to register and execute one-shot timeout handlers, while `setInterval` is used to execute a function periodically. Timeouts are registered with the `window` object that represents the window or frame that contains the document. For example, the following code snippet prints “timeout” in 300 ms after the button is clicked:

Listing 2.6: Example JavaScript Execution by setTimeout

These APIs are heavily used for producing animations in web pages by changing the position, color, and other attributes of DOM nodes according to an animation timeline [Res08].

Namespace Management

Browsers create a global scope object for every web page they load and provide an alias for it using a special JavaScript variable called `window`. JavaScript variables and functions loaded
as part of the page are then added as properties of the `window` object by the browser. Thus all scripts within a web page share the same namespace - that of the `window` object.

Sharing a single namespace for all scripts with a web page can lead to name conflicts or *namespace pollution* if a web page includes scripts from multiple sources containing global variables and functions with identical names. Moreover, all scripts within the global namespace are treated equally while allocating resources during their execution. This limitation prevents the isolation of a resource hogging script from the rest of the web page.

**Content Scripts**

Browser extensions are software components which can be installed in the browser to add new functionality to the browser. For instance, an extension in Google Chrome can interact with a user web page using *Content Scripts* that run in the context of the web page loaded in the browser [Goo10]. Content scripts can be used to implement user specific adjustments such as increasing the size of the font for texts in a web page every time the page is loaded in the browser. Content scripts and scripts in the web page belong to different namespaces. Because of that, scripts within the web page cannot access the content scripts acting on the page, nor are the content scripts allowed to access the user script code. However, content scripts run within the same execution context or thread of the user web page. Therefore, a blocking loop in a content script can cause the web page to lose responsiveness.

### 2.2 Web 2.0 Mashups

Web 2.0 mashups embed data and code from one or more sources in a single page. Examples of such web sites include popular social networking websites such as `www.facebook.com`, `www.myspace.com`, etc., and popular travel information web sites such as `www.kayak.com`, etc. Many of these sites provide JavaScript APIs that can be used to develop new features by reusing the data maintained by the websites. Table 2.1 lists the API provided by a set of popular Web 2.0 mashups to share user data, along with their popularity calculated by `www.alexa.com` based on average daily visitors and page views over the past 3 months. The smaller the rank the more is the numbers of visitors and the popularity of the web site.

Similar JavaScript API are also being used to write *widgets*, which are small pieces of
Table 2.1: JavaScript APIs provided by mashups.

<table>
<thead>
<tr>
<th>URL</th>
<th>Alexa Rank</th>
<th>JavaScript API for sharing data</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.facebook.com">www.facebook.com</a></td>
<td>2</td>
<td>Graph API and FB JavaScript SDK</td>
</tr>
<tr>
<td><a href="http://www.myspace.com">www.myspace.com</a></td>
<td>17</td>
<td>OpenSocial API and MyOpenSpace API</td>
</tr>
<tr>
<td><a href="http://www.orkut.com">www.orkut.com</a></td>
<td>61</td>
<td>Gadgets and OpenSocial API</td>
</tr>
</tbody>
</table>

JavaScript code that can be embedded in any web site to share data back and forth with data stores maintained by the websites. For example, the “Like” button widget of Facebook enables sharing of content by a Facebook user to his friends on Facebook \[Fac10\].

Scripts in embedded features and widgets are not isolated in existing browser implementations. In our work we try to provide isolation for such content in the page so that any failures related to such content can be isolated.

### 2.3 Principles of Isolation and Resource Management

Software systems that act as platforms for the execution of other programs must be able to isolate their execution from one another. Isolation in such system ensures that an executing program does not interfere with the execution of another program. For example, an isolated program should not access and modify data allocated in memory by another program. Isolation should also ensure that a failure in one program does not cause the host system or other programs to fail.

A program also needs to communicate with the host system and other programs during its execution in order to share data. The host system must provide and protect the necessary communication channels.

The execution of a program requires resources such as CPU and memory to make progress. Resource management in the host system should be able to provide resources to all executing programs. It should also ensure that no program exhausts the system’s resources.

The following subsections discuss how these principles of isolation and resource managements apply to operating systems, language runtime systems, and web browsers.
Isolation in Operating Systems

A process is the abstraction for the isolation of execution of programs in an operating system. Operating systems isolate processes from each other by associating each process with a separate area in memory called address space and by restricting a process from accessing memory and I/O ports outside its allocated address space.

A kernel is a special program that mediates all interactions between the processes executing user programs and hardware resources like CPU and memory. Multiple processes can coexist in memory. The kernel schedules the processes to acquire and release the CPU so that all of the processes can make progress. The kernel has complete access to the system memory. Requests for memory allocation from a user process are handled by the kernel, which decides whether memory can be allocated to the process or not.

The address space mechanism also ensures that processes can communicate using shared memory. When a process wants to communicate, the memory is mapped to the communicating processes’ address spaces by the kernel.

The kernel is loaded in a protected area of physical memory so that it is not overwritten accidentally by any part of the operating system or by user processes. In many operating systems such as UNIX, a user process can execute privileged kernel code by invoking system calls to request a service performed by the kernel such as I/O, memory allocation, etc. The kernel of such a system is structured such that any failure caused by user processes in the kernel does not cause abrupt termination that endangers the integrity of the kernel itself, because such termination would also affect the integrity of other applications that share the kernel.

In such operating systems, the termination of a user process while it is executing in kernel code is deferred until the control is returned to the user code. When a process receives a termination request while executing in user mode, the kernel terminates the execution immediately and reclaims the resources used by the process.

Isolation in Language Runtime Systems

A language runtime system executes user programs written in a higher level language. The Java Virtual Machine is an example of language runtime systems. Like operating systems,
language runtime systems should provide support for isolating the execution of these programs and support the management of resources allocated by the programs. Although the type safety feature of the language is used to prevent a program from performing any arbitrary memory accesses, it does not provide safe termination guarantees to user programs or control resources allocated by them.

Some research JVMs such as KaffeOS \cite{BH00, BH05} and MVM \cite{CD01} address the isolation issues in Java. KaffeOS isolates executing programs in an abstraction called a process. KaffeOS processes maintain different modes for user and kernel code in the JVM. KaffeOS extends the protection provided by the type safety of Java by giving each process its own heap where the process allocates objects. A process is not allowed to access the heap of another process. Heaps are allocated by a system wide allocator that keeps track of the memory allocated by each process. KaffeOS embeds a CPU scheduler that ensures that every process receives its assured share of CPU time. KaffeOS takes special care to minimize the time spent on non-preemptive system sections of code by a process. It employs garbage collection per process basis instead of traditional VM-wide garbage collection so that the CPU time consumed during garbage collection can be charged to the appropriate process.

Processes in KaffeOS communicate by direct sharing of classes and objects via special areas of heap memory known as shared heaps. A process in KaffeOS can create a shared heap to communicate with other processes. KaffeOS ensures that only the communicating processes can access objects in the shared heap. In order to account for memory allocation accurately, the allocation of a shared heap is charged to all the processes that communicate using the heap.

KaffeOS detects internal exceptions and termination requests for user programs. The safe termination of a process is ensured by deferring termination when the user code is manipulating kernel data structures.

The Java Multitasking Virtual Machine (MVM) \cite{CD01}, which is adopted in commercial Java virtual machines for JavaME CLDC and CDC \cite{Mic10}, provides an isolation abstraction called \textit{Task} which is similar to a KaffeOS process. MVM replicates the non thread-safe JVM components on a per-task basis and shares the rest of the JVM among tasks so that every task has the illusion of having the complete JVM during its execution. Each task is also given a logically disjoint heap. Allocation of memory in the heap is charged to individual tasks. MVM tasks communicate using copying communication mechanisms such as sockets.
or RMI.

2.3.1 Isolation in Web Browsers

Web browsers act as hosts for the execution of scripts embedded in web pages. Web browsers implement a URL based access control policy called Same Origin Policy [Rud09, JBBM06] to limit the interaction between two web pages opened in the browser. The same origin policy prevents a script loaded from one origin from getting or setting properties of a document from another origin. However, the same origin policy focuses only on the security so that the data and code in a web page is not accessed by another page loaded in the browser. Moreover, traditional web browsers do not isolate user script execution from the browser code. As a result, a buggy script can prevent any code in the browser from making progress.

In order to isolate failures from user code, Google Chromium [RG09] implemented a solution to isolate web pages loaded in the browser in separate OS processes. In Chromium, the process that loads a web page has its own copy of all browser components, including the JavaScript engine, the DOM processing component, and the HTML rendering engine. It communicates with the privileged Browser Kernel process, which consists of the cache, network and user interface components. Chromium delegates the responsibility of resource control to the operating system. Processes run according to the operating system’s scheduling policies. The communication between different components are also delegated to the operating system’s inter-process communication mechanisms. Likewise, the safe termination of processes loading web pages is guaranteed by the termination policies enforced by OS processes.

Process-based isolation helps Chromium to improve robustness as the browser is able to make progress even if one of the browser instances crashes. Chromium also provides a task manager that displays the CPU, memory and network usage of any browser process. This helps in monitoring any web site for possible problems. Unlike monolithic browsers where memory allocated to a web page is recovered using garbage collector, the process based isolation allows Chromium to release allocated memory for a browser instance simply by discarding the entire browser process. Since web pages are separated into multiple processes, scripts belonging to multiple pages can now make concurrent progress. Since browser components are replicated per process, the dependency between processes is reduced and the inter-process communication is kept low.
Multiple processes however incur a startup latency for newly created tabs. Multiple processes also increase the aggregate memory footprint of the chromium browser compared to a monolithic browser while loading multiple web pages from different sites.

Internet Explorer 8 has also introduced architectural changes to improve the reliability, performance, and scalability of the browser. The design is called LCIE (Lightly Coupled IE) [Zei08]. In LCIE, the system portion of the browser known as the UI Frame is separated into a separate process. The web pages loaded into the browser are then grouped into separate processes according to their integrity level, which indicates the trustworthiness of the web pages. These processes are known as tab processes. The tab processes communicate with the frame process using asynchronous inter-process communication mechanisms supported by the Windows OS. If one of the tab crashes, it disables the owning process which includes all other tabs running in the process. However the frame process is isolated from such failure and the remaining tab processes are able to keep continuing their progress.

Electrolysis is the code name for Mozilla’s effort for supporting process-based isolation in Firefox. As in Chrome and LCIE, separate processes are used to display the browser UI, web content, and plug-ins to provide better application UI responsiveness, improve performance in multi-core machines and provide stability [Moz09].

2.4 CPU Scheduling

A CPU scheduler enforces CPU allocation policies for each isolated entity. In an operating system, a scheduler maintains a ready queue of processes that are ready to be executed and schedules one of the processes from the queue per CPU. The execution continues until a scheduling point is reached. Commonly used scheduling points include when the running process either

- completes execution or
- consumes a certain amount of CPU or
- blocks for I/O or
- receives an interrupt.
Scheduling algorithms define the criteria for selecting processes from the ready queue. For example, a time sharing system may use a round robin scheduling algorithm so that each ready process is scheduled for a given time quantum. The types of processes that a system supports can influence the choice of scheduling algorithm. For example, systems which support both interactive and batch processes may use separate ready queues for each type of processes and schedule processes in each queue with different scheduling algorithms.

**Proportional Share Schedulers**

In proportional share scheduling, each client process receives CPU time according to its assigned weight. There are many variations of proportional share schedulers [DC99, DKS89, NVZ01, KL88, WW95]. Accuracy of resource allocation and low scheduling overhead are desirable properties of a proportional share scheduler. *Fairness* and *Service Error* are measures of accuracy in CPU allocation [NVZ01].

The service error $E_A(t_1, t_2)$ for client $A$ over interval $(t_1, t_2)$ is the difference between the actual time, $W_A(t_1, t_2)$, allocated to $A$ during $(t_1, t_2)$ and the ideal allocated time proportional to its associated share, $S_A$. The service error is calculated as:

$$E_A(t_1, t_2) = W_A(t_1, t_2) - (t_2 - t_1) \frac{S_A}{\sum_i S_i}$$

A system is said to achieve perfect fairness when its service error is zero. In real systems, it is not possible to achieve perfect fairness. A good proportional share scheduling algorithm has a service error close to zero. A positive service error indicates that the process is receiving more CPU than it should. A negative service error indicates that the process is receiving less CPU than the amount that would be allocated under ideal conditions.

In traditional proportional share schedulers, threads take turns to acquire the CPU according to their weights. This scheme creates problems while supporting latency sensitive threads such as threads performing IO. Such threads wake up for a short period of time, process the available data, and go back to sleep again. In the browser, UI event handling threads are examples of such latency sensitive threads as they must process events on time to provide feedback to the user interacting with the browser.
Borrowed Virtual Time Scheduler

The Borrowed Virtual Time Scheduler (BVTS) [DC99] is a proportional share scheduler designed to provide low latency for real-time and interactive applications while maintaining weighted sharing across applications in the system.

BVTS uses a measure called virtual time for each client to record how much of its proportional CPU allocation a client has received relative to other clients. When a client executes, the scheduler accounts for its running time in units of *minimum charging unit* or *mcu* and advances the *actual* virtual time of the thread by dividing the running time by the CPU share of the thread. However, the scheduler schedules the runnable thread with the earliest *effective* virtual time.

The effective virtual time $E_i$ of a thread $i$ is calculated as

$$E_i = A_i - (\text{warp} \ ? \ Warp_i : 0) \quad (2.1)$$

where $A_i$ is the actual virtual time of the thread, $Warp_i$ is the virtual time warp of the thread and *warp* is a boolean set to true for latency-sensitive threads. In BVTS, all runnable threads consume the same quantum of virtual time over a period, thus ensuring fairness. In order to schedule a latency-sensitive task immediately, BVTS warps back a latency-sensitive thread in virtual time as it wakes up, making its effective virtual time earlier than those of the other runnable threads.

To not violate asymptotic fairness, there is a limit on how long a thread can run warped, denoted by a parameter, *WarpLimit*. This limit prevents a condition where a thread with a large warp would otherwise hog the CPU by virtue of its low effective virtual time. Another limit called *UnwarpLimit* can be set to prevent a thread from receiving warps if the thread wakes up more than once within its UnwarpLimit.

At each clock interrupt the virtual time of the running thread $i$ is updated. A context switch takes place if there is a runnable thread $j$ such that

$$A_j \leq A_i - \frac{C}{w_i} \quad (2.2)$$

where $C$ is a constant called *context switch allowance* that ensures that two threads with equal virtual times do not context switch at every clock interrupt.

The scheduler maintains a scheduler variable called *scheduler virtual time*, which is equal to the minimum virtual time of any runnable thread. When a thread $i$ wakes up after sleeping
a long time, its virtual time is set to SVT to prevent the thread from claiming excessive amount of CPU to make up for the period during which it was asleep.

The following example illustrates the BVTS algorithm:

Let us consider a system with four threads A, B, C and D using a BVT scheduler with assigned weights 10, 20, 10 and 60. Among the threads, C is a latency sensitive thread that is processing UI events. C wakes up after 5 mcus. It then uses 5 mcu of CPU time and goes back to sleep. It has the following warp values: warp, $W_C = 50$ in virtual time units; warpLimit, $L_C = 5$ mcu and unwarpLimit, $U_C = 10$ mcu. A and B are CPU bound threads, which stay active from 0 mcu to 16 mcu. Thread D wakes up after a long sleep period after 12 mcus. We assume that the context switch allowance is 2 mcu for the system.

Figure 2.3: BVTS example. The X-axis represents the time in mcu and Y-axis represents the effective virtual time of the threads A, B, C and D.

Figure 2.3 shows how these four threads will be allocated CPU time during the first 16 mcu. Thread A is scheduled first. It runs till the 2nd mcu when the difference of the virtual time of
the thread $A$ and the context switch allowance in virtual time becomes equal to the virtual
time of the thread $B$. A context switch takes place according to equation $2.2$. At the $5^{th}$
clock interrupt, $C$ receives an UI event and wakes up. The thread warps back in virtual time
because the UI event is latency sensitive. The virtual time of $C$ becomes

$$SVT - W_C = 6 - 50 = -44$$

Thus $C$ runs for 5 mcu until the $9^{th}$ mcu. At this point, the scheduler selects thread $B$ with
the earliest effective virtual time. The threads $B$ and $A$ alternate use of CPU until $D$ wakes
up. $D$’s virtual time is updated to 12 based on SVT value. From that point on, the three
threads $A$, $B$ and $D$ take turns according to their weights.

### 2.5 Models for Concurrency in Software Systems

ScriptSpaces support concurrent execution of multiple contexts. Software systems use either
thread-based models, event-based models, or hybrid methods to achieve such concurrency.

#### 2.5.1 Event-Based Concurrency

In an event-based architecture, the request processing logic is broken down into multiple
stages; each stage is responsible for some part of request processing. A request may be
a client request in the case of web servers or a UI event in a graphical application. The
input to a stage is an event that encapsulates all the information necessary to process the
request. Event processing at each stage continues until it reaches a point when some blocking
operation is needed. At this point the execution yields so that a scheduler can schedule
other events associated with a different request. When the blocking operation completes,
a continuation event is generated from the data passed to the blocking operation. The
continuation event is then scheduled by the event scheduler of the system. Concurrency is
thus achieved by interleaving stages from multiple requests.

Event-based concurrent systems, however, require extra development effort to schedule events
efficiently in multiple stages for different requests. In addition, the complete processing in-
formation for the next event must be saved before a stage can yield its execution. This
process, also known as stack ripping, is tedious. An implementation of event-based concurrency called Tame $[KKK07]$ uses source to source translation techniques to generate stack
ripping code automatically.

2.5.2 Thread-Based Concurrency

Thread-based concurrency is achieved by offloading every request to a separate processing thread. A scheduler then schedules each of the threads, thus providing concurrent execution. Stack ripping is not required when an execution makes a blocking call and resumes execution because the thread’s stack contains all necessary context. However, the developer must synchronize access to shared data while writing the request processing logic, which is prone to errors and limits scalability. Some implementations limit the maximum numbers of threads that can be created and use a queue to hold pending requests if that maximum is reached.

2.5.3 Hybrid Approaches for Concurrency

SEDA [WCB01] exploits both event-based and thread-based concurrency by providing an event handler, an incoming event queue, and a thread pool to each stage of request processing. A controller component controls scheduling and thread allocations for each stage. Within a stage, the threads retrieve multiple events from the event queue of that stage and execute the event handlers on them. Subsequently the stage may dispatch one or more events by enqueuing them on the event queues of the next stage for further processing. SEDA optimizes the throughput and response time by batching events and smartly choosing adaptive algorithms for tuning thread pool sizes.

2.5.4 Migrating Thread Model

Handling of a client request may span multiple stages or protection domains and may have multiple threads associated with them. The motivation for migrating threads [FL94] is to allow a single thread to cross multiple stages or protection domains to process the request.

The migrating thread model decouples the thread abstraction into two parts - first, a logical flow of control, represented as a stack of execution contexts generated from the code in multiple stages that represents the complete request processing, and second, as the schedulable entity with priority and other resource control attributes. Such a system provides advantages over systems that have a static association of threads to stages in request processing.
It reduces costly thread-to-thread context switches since the request processing moves from one stage to another in a single thread.

Migrating threads are used in Mach 3.0 operating systems to optimize remote procedure call or RPC. Such calls comprise a client task that originates the call and a server task that provides the service for the call. RPC is used extensively within the Mach OS for communication between user tasks and components in the Mach kernel. In RPC, the flow of control moves between the server and the client tasks and back. The use of migrating threads enables each RPC call to become a single identifiable entity to the kernel by allowing a single thread to move from one task to another. The kernel now does not wake up a server thread on a request from the client task, but allows the client thread to continue executing the server code requiring no rescheduling or full context switch. In addition, the resource accounting for RPC is improved as the work done on the server on behalf of the client can now be automatically accounted.
The chapter discusses the design rationale for ScriptSpaces. We define the ScriptSpace abstraction and discuss how it is used to provide isolation in section 3.1. We discuss the user/kernel boundary for ensuring safe termination of script execution in the browser in section 3.2. In section 3.3, we discuss how a migrating thread model can be used to implement concurrent script execution in a browser that supports ScriptSpaces. We discuss the design decisions for supporting resource control in section 3.4.

3.1 Isolation using ScriptSpaces

ScriptSpace is an abstraction that provides separate, isolated execution environments for parts or all of a web page in a web browser. We divide the scripts in web pages into disjoint groups containing one or more scripts and tie their execution to a ScriptSpace. Within each ScriptSpace, a single-threaded JavaScript execution environment is maintained. Multiple ScriptSpaces within a page are isolated with respect to namespace, CPU, and memory consumption. Multiple threads can coexist in the browser, executing scripts that belong to different ScriptSpaces.

3.1.1 ScriptSpace Association

Web designers use HTML container elements such as div, frame and iframe to group and organize content within a web page. In a conventional browser, this grouping does not extend to scripts when the page is loaded because the browser conflates all scripts in a
single namespace. By associating the set of scripts within an HTML container with a set of ScriptSpaces, we extend the design time grouping of content to their associated scripts.

A ScriptSpace is created by specifying a dedicated ScriptSpace attribute for the `body`, `div`, `frame` and `iframe` HTML container elements in a web page. When loading a page, the browser looks for this attribute, creates a ScriptSpace and associates all scripts referenced by or embedded in descendants of the container element with the ScriptSpace. Most web sites, including mashups, integrate third-party content in separate containers. Mapping ScriptSpaces to subtrees of the DOM tree in this way simplifies the grouping of third-party scripts, as the only change required to group is to add a ScriptSpace attribute to the container element.

In the absence of any ScriptSpace denominations, the browser creates a default ScriptSpace for a web page, as well as for any frames and iframes within the page. It ensures that two web pages loaded in two different tabs are isolated. Thus existing web sites will be able to achieve page-level isolation without any changes to their code. Similarly, mashups that embed third-party content in iframes will benefit from the isolation without any changes to the code. If scripts are loaded before any of the ScriptSpace attributes is encountered, they are associated with the default ScriptSpace of the web page. For example, the browser recognizes the class attributes `SS_1` and `SS_2` to be ScriptSpaces for the code snippet 3.1 and associates the event handlers in the `div` elements `clickHandlerSS1` and `clickHandlerSS2` with separate ScriptSpaces. If the ScriptSpaces were not present, the browser would have associated the event handlers with the default ScriptSpace. Figure 3.1 illustrates the web content to ScriptSpace mapping for the code snippet 3.1.

```
<body>
  <div class="SS_1">
    <input type="button" onclick="clickHandlerSS1();" value="Click Me!" />
    <div id="A"> Counter </div>
  </div>
  <div class="SS_2">
    <input type="button" onclick="clickHandlerSS2();" value="Click Me!" />
    <div id="B"> Counter </div>
  </div>
</html>
```

Listing 3.1: Example of ScriptSpace association

In some situations, an executing script may register new event handlers with the nodes of
Figure 3.1: Example of ScriptSpace association. ScriptSpaces SS_1 and SS_2 are mapped to the two top level div elements as shown. The browser associates the scripts embedded in and referenced by the two divs and their descendants to the corresponding ScriptSpaces so that their execution can be isolated.

the DOM tree. In such cases, the newly registered event handlers are associated with the ScriptSpace of the executing script so that the event handlers are executed on behalf of the ScriptSpace that has registered them.

3.2 User/Kernel Boundary in the Browser

In a browser, scripts may make calls into native browser components. Since these browser components are shared among all script executions in the browser, we do not allow abrupt termination of a script while executing in such components. Instead, we wait till the execu-
tion reaches a safe-point in the user code and then terminate the script. In our model, we use separate user mode and kernel mode as a means to enforce safe termination of scripts in the browser by gracefully handling any termination condition in the browser code that may be generated from operations like failed resource allocation attempts, or by a user request. In our model, scripts are instrumented to insert a callback which is invoked after every branch instruction in the original script. The callback checks for the termination condition, and quits the execution if indicated.

![Figure 3.2: User/Kernel boundary in web browsers. The kernel consists of the native implementation code of the browser. Scripts embedded in web pages and in the browser extensions are considered to be user code. The resulting user/kernel boundary ensures that termination of script execution is safe in the browser.](image)

Some browsers support extensions written in JavaScript. In our model, scripts in extensions are associated with ScriptSpaces of a special type, called trusted ScriptSpaces so that they are distinguished from the user scripts embedded in the web pages. All user scripts loaded from a web page are associated with ScriptSpaces with an untrusted type. Since the extension scripts also access the browser’s critical components during their execution, we defer termination of the execution to safe-points as we do for user scripts embedded in web pages. Figure 3.2 illustrates the user/kernel boundary that enforces safe termination.
3.3 Concurrency Model for ScriptSpaces

We considered two different approaches for providing concurrent execution of scripts in the browser based on the isolation provided by ScriptSpaces. The first attempt was an event-based concurrency model. We discuss the first model and its limitations in order to highlight the benefits of the final model we chose, which is based on migrating threads.

3.3.1 Event-Based Approach

In this approach, we statically assigned a thread to a ScriptSpace to execute the scripts associated with it. During DOM event handling, a ScriptSpace acted as a stage for event handling by executing an event handler in its thread, and creating and passing a continuation event object to the next ScriptSpace to execute the next handler according to the event flow. Continuation events were added to a queue in the ScriptSpace, which was polled by each ScriptSpace’s thread. This design allowed stages to run in parallel, enabling handling of more than one event within the browser.

Figure 3.3 illustrates an example of event handling using this model when a user clicks on a button in the web page shown in figure 3.1. We assume that for the click event, an event handler at the container div element is executed during the capturing phase of the flow. The click handler of the input button is executed next and finally one event handler each at the container div and body element is executed during the bubbling phase of the event processing.

However, being an event-based model, the above approach required manual stack ripping for each UI event that was being dispatched to the ScriptSpaces. In order to propagate the resulting DOM event through appropriate ScriptSpaces, the continuation event object consisted of a list of remaining event handlers, prepared by traversing the DOM nodes of the web page in the browser according to DOM event flow model, and the execution stack required to continue after the execution of an event handler completed.

In a browser, an UI event is passed through multiple stages before it is converted to a DOM event and the actual event handlers are executed. For example, the browser receives the mouse click event as an object containing the location of the mouse pointer in the display and a type indicating that it is a click event. The browser then maps the point to the document, finding the HTML element in the document the point corresponds to, and
Figure 3.3: Event-based concurrency. A click on the button will result in the DOM event flow as shown. After pre-processing, which converts the UI event to DOM event, the DOM event is offloaded as a continuation event to the ScriptSpace SS_1. The event handlers at the div and input button elements are executed by the thread associated with the ScriptSpace. Then a new continuation event is created and dispatched to Default ScriptSpace so that the event handler of the body element is executed, as it is associated with the Default ScriptSpace. Finally, the post-processing logic is executed based on the result of the DOM event handling.

creates and dispatches the ‘click’ DOM event. The result of event handling is propagated back through multiple layers within the browser. This nesting of several components made stack ripping for events tedious. We realized the limitation of our design, and chose a model of concurrency that is based on a thread-based model which does not need stack ripping.
3.3.2 Migrating Threads

In our final design, we use a migrating thread model for event handling in the browser to accommodate event handlers associated with different ScriptSpaces. In our model, a migrating thread moves from one ScriptSpace to another, executing event handlers at each ScriptSpace during event handling. Using one thread for the processing of an entire event simplifies maintaining the context of each event, thus ensuring the ordering of DOM event handling without any stack ripping overhead. Figure 3.4 illustrates the processing of the click event from example 3.3 using a migrating thread.

Figure 3.4: Migrating thread for concurrency. The event flow for the click event completes within the same migrating thread. While the thread executes the event handlers at the div and the input button, it uses the resource control attributes of ScriptSpace SS.1. When the thread migrates to the default ScriptSpace to execute the event handler at the body element, the thread uses attributes of the default ScriptSpace for resource allocation.
The execution makes progress according to the CPU and memory available to the associated ScriptSpace. Within each ScriptSpace the resource accounting attributes, such as CPU shares, CPU consumption, etc., of the thread are set to the attributes associated with the ScriptSpace. Figure 3.4 illustrates how the event flow and resource control decision changes as the thread migrates from one ScriptSpace to another.

The threading model is also adaptable to other types of script execution in the browser. For inline scripts, the thread that interprets the script content within the `script` element acts as the migrating thread, migrating to different ScriptSpaces when required. Timeouts and XHR responses cause a single script execution which progresses in an event handling thread with the resource control attributes of the associated ScriptSpace.

Since JavaScript does not provide any thread safety, the migrating thread acquires exclusive ownership of the ScriptSpace before executing any event handler. Exclusive ownership ensures that the scripts within a ScriptSpace are executed one at a time, creating a single threaded execution environment within the ScriptSpace. When multiple threads try to acquire the ScriptSpace at the same time, only one thread obtains the ownership of the ScriptSpace and the rest of the threads wait. Since events are latency sensitive, we limit the numbers of threads that can wait on a ScriptSpace and reject the event handling requests beyond that limit. Recurring events such as timeouts are rejected if the ScriptSpace cannot be acquired on the first attempt. This policy prevents such events from filling up the wait queue.

### 3.4 Resource Management

ScriptSpaces are used as resource principals to provide resources to a script execution. In order to provide low latency for UI events, even in the presence of a CPU bound script execution, we use a proportional share scheduler based on the BVT scheduling algorithm for the management of the CPU time allocated to each ScriptSpace.

A ScriptSpace contains a CPU weight parameter that is used by the BVT scheduler to allocate CPU time to the execution of scripts associated with the ScriptSpace. Since the BVT scheduler is work conserving, it allows scripts to utilize unused CPU time when no other execution is in progress. When multiple ScriptSpaces contend for the CPU, the browser shares the available CPU time among the executions proportional to their CPU share parameters.
A ScriptSpace records the CPU time consumed by the execution of its associated scripts, which is used as a criteria for yielding the current execution to another execution after the ScriptSpace consumes CPU time proportional to its CPU share. The other BVT scheduler parameters, i.e., warp and warp limit, are used to provide momentary scheduling advantages to a particular ScriptSpace. For example, trusted ScriptSpaces are given a large warp and warp limit value so that they execute scripts to completion without having to yield to other scripts.

The ScriptSpace model could also be extended to support the control of memory allocation by specifying a limit on the memory allocated during the execution of a script. The memory limit of the ScriptSpace could be verified each time some memory is allocated. When the limit is reached, scripts associated with the ScriptSpace could be denied any new memory allocations.

Implementing resource control within the browser provides opportunities to develop tools to monitor resource consumption by the end users. A set of APIs can be provided by the implementation to control resource allocations by a user so that the user can penalize a faulty script execution by removing resources from its associated ScriptSpace or prioritize an execution by increasing the resource limits for its ScriptSpace.
Chapter 4

Implementation

We have implemented the ScriptSpace prototype by extending the Firefox v3.0b2 codebase. We chose Firefox as our testbed because it is an open source, mature web browser with support for popular web standards, including DOM, HTML, CSS, and ECMAScript.

We present an overview of the changes made to the Firefox codebase in section 4.1. We discuss the details of our changes to the Firefox codebase in subsequent sections. The changes made within Firefox to allow concurrent event processing are discussed in section 4.2. We discuss the ScriptSpace implementation in section 4.3. Our implementation supports the management of CPU resources, which is discussed in section 4.5. We conclude the chapter with a discussion of the SSManager UI tool we implemented for monitoring CPU usage by executing scripts.

4.1 Overview

Figure 4.1 shows the changes we made in Firefox for supporting ScriptSpaces.

The browser consists of three major modules -

1. Networking that retrieves web pages and provides support for data caching and asynchronous I/O.

2. Layout that parses HTML, creates the DOM tree, renders the web page in the browser and provides support for user interaction with the rendered web page.
3. *Script Execution* which provides an environment to execute scripts in the browser and ensures safe access to content by script code.

The *Base Infrastructure Components* provide the necessary framework for implementing the above modules in the browser.

Figure 4.1: Overview of changes made to Firefox. We have added the ScriptSpace Management and the BVT Scheduler components to the existing Firefox architecture.

There were no changes to the networking module. We made major changes to the *event processing* component of the layout module to support concurrent script execution. We modified the rendering component to enable rendering of the UI from event handling threads. The XML parser was modified to recognize the ScriptSpace keywords. The content model was modified to add a ScriptSpace as an additional attribute for an event handler during registration. We introduced two new components to the script execution module: *ScriptSpace*
Management, which implements the ScriptSpace abstraction and manages the ScriptSpace instances, and BVT Scheduler, which implements the borrowed virtual time scheduler to schedule script executions. Additional changes were required to the JavaScript engine in Firefox to support entering and leaving the scheduler while a thread is executing in the engine.

The XPCOM (Cross Platform Common Object Model) component provides support for code written in different languages like C++, JavaScript, Java, etc., to coexist within the same environment. It also provides core functionality such as file and memory management, threads, etc. We have modified the event handler thread implementation to invoke the scheduler before blocking on the event queue and delay event processing until after selected by the scheduler.

XPConnect is the component that provides the JavaScript to C++ language binding and loads XPCOM code written in JavaScript. We have modified the XPConnect component to add ScriptSpace information to the scripts loaded by XPConnect. We did not change the NSPR (Netscape Portable Runtime) component, which provides a device independent API for system level functions.

Our changes to the Firefox codebase include 258 file changes (approximately 1.5% of the total files), the addition of 15357 new lines of code and the deletion of 396 lines of existing code. Compared with the size of the Firefox codebase, which is approximately 5 million lines of code, this change looks reasonable. We believe that a design like ours could easily be integrated with other browsers which share the same standard browser interfaces provided by DOM, HTML, JavaScript, and other standards as found in Firefox.

### 4.2 Concurrent Script Execution

Firefox uses a single threaded execution model for handling user events. The main event loop polls two different event queues for events. User events are added to the first queue maintained by the underlying UI toolkit, e.g., Gtk [Kra07] in the Linux version of Firefox. The main thread retrieves all UI events that are accumulated in the queue between two poll operations, and executes them one by one. Before the thread completes the loop, it polls the second event queue. The second queue is used to manage events internally generated by Firefox. The internal events are known as nsRunnable events as they are processed by
calling their `Run` method. These events are generated during UI event processing to execute some part of the processing logic, as well as by the network thread to notify the main thread when network data was received, and by the timer thread when a timer expires. The main loop checks the state of the browser before it makes another iteration. Unless the processing of an event sets the browser state to “shutting down,” the main event loop continues as shown in the flow chart.

![Flow Chart](image)

**Figure 4.2:** Main event loop in Firefox. The main event loop in Firefox retrieves UI events and events internal to Firefox and processes them within the same thread.

In order to prevent a malicious script execution from blocking the main event loop, the event processing model in Firefox is split into two parts - event retrieval and event processing, and any processing of events that causes the execution of scripts is no longer performed in the main thread. The main thread still retrieves UI events from the queue, but instead of processing the events right away, it adds the events to another queue as shown in figure.
Internal events such as timeouts and network responses which may invoke JavaScript code are added to this queue instead of the internal event queue of the main thread. Internal events that do not cause the execution of any user scripts continue to be processed by the main thread.

Figure 4.3: Processing of script invoking events in modified Firefox. The main event loop still retrieves the UI events but instead of processing them, it offloads them to another event queue. In addition, some of the internal events generated by the main thread, by the network thread, and by the timer thread, which cause script execution, are added to the queue as shown. These events are processed by an event handler thread pool.

We use a thread pool configured with a fixed number of threads for retrieving events from the queue. After a thread removes an event from the queue, the thread continues processing the event by executing all the event handlers. Since the handling of an event is short lived in the common case, a fixed thread pool suffices for our purposes.
4.3 ScriptSpace Management and Association

Our implementation for ScriptSpace management consists of two interfaces - \textit{nsIScriptSpace}, which encapsulates various ScriptSpace facilities, and \textit{nsIScriptSpaceRuntime}, which facilitates the management of ScriptSpace instances created in the browser.

The public methods and the attributes of the ScriptSpace component are listed in table 4.1. In order to provide a single threaded JavaScript execution environment, our implementation maintains a JavaScript context object, which provides the JavaScript engine's execution stack. Although the context can be used for multiple invocations, only one thread is allowed to execute scripts in the context at a time. Whereas the original Firefox code created a single JavaScript context for each HTML document loaded in the browser, we create one JavaScript context per ScriptSpace so that more than one contexts can now exist for a web page loaded in the browser.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ScriptSpace general attributes</strong></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>type of the ScriptSpace, e.g \texttt{trusted}, \texttt{untrusted}</td>
</tr>
<tr>
<td>name</td>
<td>human readable name of the ScriptSpace</td>
</tr>
<tr>
<td>queueLength</td>
<td>the maximum numbers of threads that can wait on the ScriptSpace for ownership</td>
</tr>
<tr>
<td><strong>Attributes used for CPU resource control</strong></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>CPU share</td>
</tr>
<tr>
<td>warpLimit</td>
<td>warp limit</td>
</tr>
<tr>
<td>unwarpLimit</td>
<td>unwarp limit</td>
</tr>
<tr>
<td>virtualTime</td>
<td>virtual time</td>
</tr>
<tr>
<td>totalExecutionTime</td>
<td>total execution time in usec</td>
</tr>
<tr>
<td><strong>Public methods</strong></td>
<td></td>
</tr>
<tr>
<td>init</td>
<td>initializes attributes</td>
</tr>
<tr>
<td>acquireOwnership</td>
<td>obtains exclusive ownership of the ScriptSpace</td>
</tr>
<tr>
<td>releaseOwnership</td>
<td>releases ownership of the ScriptSpace</td>
</tr>
<tr>
<td>alreadyOwned</td>
<td>checks if the ScriptSpace is already owned</td>
</tr>
<tr>
<td>kill</td>
<td>request termination of the ScriptSpace</td>
</tr>
<tr>
<td>isKilled</td>
<td>checks if a termination request was received</td>
</tr>
</tbody>
</table>

An \textit{nsIScriptSpaceRuntime} object keeps a record of the created ScriptSpaces per document.
as well as the active ScriptSpaces which are currently executing scripts. A single instance of nsIScriptSpaceRuntime is created for the browser. Table 4.2 lists the public methods of the interface.

The nsIScriptSpaceRuntime object maintains a stack of currently owned ScriptSpaces per thread of execution. The getThreadScriptSpaceEnumerator method provides an iterator to iterate over all ScriptSpaces in the stack which can be used for retrieving the resource consumption statistics of the ScriptSpaces. The getScriptSpaceEnumeratorForDoc method can be used to traverse all ScriptSpaces associated with an HTML document loaded in the browser. An example usage is shown in code snippet 4.1 which retrieves the total execution time for all ScriptSpaces created for the HTML document.

```
1 // obtain a reference to the nsIScriptSpaceRuntime service
2 var ssRuntime = Components.classes["@sl.cs.vt.edu/ssruntime"].
   getService(Components.interfaces.nsIScriptSpaceRuntime);
3
4 // obtain iterator over all ScriptSpaces for HTML document ddoc
5 var ssEnumerator = ssRuntime.getScriptSpaceEnumeratorForDoc(ddoc);
6
7 // traverse all ScriptSpaces and print the total execution time
8 while( ssEnumerator.hasMoreElements()){  
9   var n = ssEnumerator.getNext();
10   var ss = n.QueryInterface(Components.interfaces.nsIScriptSpace);
11   dump("total execution time for "+ ss +" is :"+ ss.totalExecutionTime+" 
12     \n");
13 }
```

Listing 4.1: Retrieving ScriptSpace information using API provided by nsIScriptSpaceRuntime

In addition, the nsIScriptSpaceRuntime instance is configured to identify a set of ad-hoc keywords like SS_1, SS_2, etc., as ScriptSpace identifiers, so that ScriptSpaces can be created when such keywords are encountered as CSS class attributes. The instance is also configured to set up initial resource control parameters and the type for the ScriptSpaces.

**Association**

Firefox internally represents DOM event handlers as instances of the nsListenerStruct type. We have modified the nsListenerStruct type to maintain the ScriptSpace information for the event handler. Similar changes were made to the data structures that represent a timeout event handler and the response handler for an XMLHttpRequest. The ScriptSpace
Table 4.2: nsIScriptSpaceRuntime API

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getThreadScriptSpaceEnumerator</td>
<td>retrieves an enumerator on all the ScriptSpaces owned by the current thread</td>
</tr>
<tr>
<td>getActiveScriptSpace</td>
<td>returns the most recently owned ScriptSpace by the current thread</td>
</tr>
<tr>
<td>getScriptSpaceEnumeratorForDoc</td>
<td>retrieves an enumerator for all the ScriptSpaces created for an HTML document</td>
</tr>
</tbody>
</table>

field is populated when an event handler is registered with a node of the DOM tree of the document.

The associated ScriptSpace is retrieved from the event handler object during the execution of the handler and acquired by invoking the acquireOwnership method of the nsIScriptSpace interface. By default, a thread blocks if the ScriptSpace is already acquired by another thread. However a flag can be passed as a parameter to the method to request an immediate return from the call if the ScriptSpace is already owned by some other execution. A typical event handler execution is shown in example 4.2:

```c
PRBool owned, released;
nsCOMPtr<nsIScriptSpace> ss(handler->mSS);
// own the ScriptSpace associated with the handler
if(NS_FAILED(ss->AcquireOwnership(PR_FALSE, &owned)) || (owned == PR_FALSE)) {
  // not able to own the ScriptSpace; returning failure
  return NS_ERROR_FAILURE;
}
// execute event handling code, e.g.
PRBool handled = handler->HandleEvent();
// release the ownership
if(NS_FAILED(ss->ReleaseOwnership(&released)) || (released == PR_FALSE)) {
  return NS_ERROR_FAILURE;
}
return handled;
```

Listing 4.2: An event handler execution using ScriptSpace. The ScriptSpace is acquired exclusively before the execution of the handler.
The acquired ScriptSpace is recorded as active ScriptSpace in a thread-local variable and the script is executed using the JavaScript context associated with the ScriptSpace. If the executing script registers any new event handlers to some node, the ScriptSpace of the executing handler is used as the associated ScriptSpace of the newly registered handlers. Inline scripts, which are executed only once during the document load, acquire the ScriptSpace of the container node they are embedded in. If no such container explicitly embeds the inline scripts, the scripts inherit the default ScriptSpace of the document.

4.4 Implementing Termination

Termination requests in the browser are generated when a user closes a tab or browser window. We propagate the termination request to all ScriptSpaces belonging to the closing document by invoking the kill method on each ScriptSpace before the document is destroyed. The kill operation sets a flag within the ScriptSpace. The flag is checked from an active ScriptSpace every time a branch instruction in the script is executed via a callback registered with the JavaScript context. If the flag is set, the execution is terminated by returning an error condition from the callback.

4.5 CPU Scheduling within Firefox

We have implemented a user level thread scheduler using the BVT scheduling algorithm to schedule script executions. The scheduler maintains different ready queues for three priority levels: HIGHEST, HIGH, and NORMAL in decreasing order of priority. The main thread is given the HIGHEST priority when it is ready to process internal Firefox events. The HIGH and NORMAL priorities are used for the event handling threads of the thread pool. A variable running thread is used to record the scheduled thread.

In our implementation, a thread calls the SL_SchedulerEntry method of the scheduler interface to schedule itself as soon as it is ready to process an event. With the highest priority, the main thread can complete event processing and resume polling for new events as soon as possible. An event handling thread is given a boost by setting its priority to HIGH before the first attempt at scheduling, so that it can make progress and acquire the associated ScriptSpace of the first event handler. Once the ScriptSpace is acquired, the priority of the
Figure 4.4: Flow chart for scheduler entry. A thread which is ready to process an event registers itself with the scheduler for scheduling.

thread is reset to NORMAL and the thread is re-scheduled with the resource control parameters of the ScriptSpace. The threads waiting in the queue of HIGH are scheduled in FIFO order. Within the queue of NORMAL priority the scheduler always schedules the ready thread with the “earliest” effective virtual time. Flow chart 4.4 illustrates how a thread is scheduled in this queue.

When the event handling thread acquires ownership of new ScriptSpace the virtual time consumed by the ScriptSpace becomes the virtual time of the thread for scheduling purposes. At each release of ownership, the virtual time of the ScriptSpace is updated, and the virtual time of the thread is reset to the value of the previous ScriptSpace, if any. The virtual time of the ScriptSpace is incremented each time the branch callback of the JavaScript context is invoked. The running thread yields the scheduler from the branch callback call-
Figure 4.5: Flow chart for scheduler exit. The thread exiting the scheduler schedules the next ready thread in our implementation.

An execution may block while trying to acquire shared resources such as the global lock of the garbage collector in the JavaScript engine. If a thread blocks for any resource during event handling or completes its execution, the thread calls the SL_SchedulerExit method of the scheduler and removes itself from the scheduler before it blocks. As shown in flow chart 4.5, the executing thread schedules another thread before it exits the scheduler. As the thread unblocks, the thread tries to re-enter the scheduler with NORMAL priority, its virtual time set to the scheduler virtual time (SVT) minus a warp boost.
4.6 SSManager UI

The SSManager UI is a Firefox extension developed using the API provided by the ScriptSpace management component to monitor the CPU usage by different ScriptSpaces in web pages. The UI also allows the user to change the resource control attributes of ScriptSpaces.

The interface is divided into three components:

- ScriptSpace Navigator
- Visualizer
- ScriptSpace Controller

A screen-shot of the SSManager v1.0 is shown in figure 4.6.

ScriptSpace Navigator

The ScriptSpace Navigator can be used to navigate to different ScriptSpaces associated with the web pages loaded in the browser. It has a tree-like structure with the leaf nodes representing the ScriptSpaces and the non-leaf nodes representing either the web page or the frames within a web page. Leaf nodes are named after the name attribute of the ScriptSpace. Top-level nodes are named Tab[i], where i is an integer assigned to each open tab in the browser. The title or the URL of a page is not used for naming the nodes as titles and URLs may be very long and the user may have multiple copies of the same page opened in different tabs. Intermediate non-leaf nodes represent frames within a page and are labeled as Frame with an integer identifier.

Visualizer

The Visualizer component provides a real-time visualization of the CPU usage of the active node, which is the currently selected Tab or Frame node in the ScriptSpace Navigator. A script periodically polls the ScriptSpaces of the active node for the total CPU usage at the
end of the period. The default length of period is 100msec. These data is then converted to percentage CPU usage by the ScriptSpace at the $n^{th}$ period by using the following equation

$$\%\text{CPU usage} = \frac{\text{total CPU usage } n^{th} - \text{total CPU usage } (n - 1)^{th}}{\text{length of period}} \times 100$$  \hspace{1cm} (4.1)

The visualizer displays 50 such values by dynamically ordering them with respect to the sampling interval along the horizontal axis. Different colors are used to plot these graphs for different ScriptSpaces within the page, with the currently selected shown in white. The user can compare the CPU consumption of the selected ScriptSpace against other ScriptSpaces in the page while changing different parameters of the selected ScriptSpace.
**ScriptSpace Controller**

The ScriptSpace Controller interface allows the user to change the parameters associated with a selected ScriptSpace. The current values of weight, warp, warp limit, unwarp limit, virtual time and total execution time of the ScriptSpace are displayed on a form. The weight and warp related parameters can be changed in the form.

The SSManager extension is used as an example client of the ScriptSpace management API, on top of which more sophisticated, user friendly applications could be developed. For example, a manager application could remove the low level details of the scheduling parameters and provide a simple knob-like element to tune the CPU usage of a ScriptSpace. The API could also be used to create defensive tools for DOS attacks aimed at computational resources, which would auto detect the CPU usage of a ScriptSpace and automatically change its resource control parameters according to a policy.
Chapter 5

Experimental Evaluation

We evaluated the effectiveness of ScriptSpaces for isolation. In section 5.1, we discuss our experiments to show that ScriptSpaces isolate script content as specified by the developer and control the CPU allocation of each isolated ScriptSpace. We discuss our evaluation of the improved robustness in ScriptSpace enhanced Firefox in section 5.2.

We performed the experiments described in this chapter on a computer running Fedora Core (Linux kernel 2.6.26.8-57.fc8) with a Intel Pentium Dual CPU T2330 1.60GHz processor and 2GB of main memory.

5.1 Isolation Goals

Isolation of Script Content

We allow a developer to isolate the script content in web pages by associating the scripts with appropriate ScriptSpaces. In this experiment, we show that our implementation is able to maintain the association by providing a single threaded execution environment within a ScriptSpace and by allowing multiple threads to execute scripts associated with different ScriptSpaces concurrently.

The scripts in this experiment create a loop that increments an integer to 10000, log the timestamp, and reset the integer to zero several times. We used the scripts as click handlers for two buttons included in separate div elements. For the first web page, we did not specify any ScriptSpace so that both of the click handlers were associated with the default
ScriptSpace of the web page. We specified two different ScriptSpaces, $SS_1$ and $SS_2$ as the CSS class attributes of the div elements in the second web page so that each of the two click handlers was associated with a different ScriptSpace.

We obtained the graphs 5.1 and 5.2 for the web pages after plotting the number of iterations completed by the scripts against the execution time.

![Graph 5.1: Progress of scripts associated with a single ScriptSpace. The plot was obtained by recording the number of completed counts to 10000 by each click handler vs. the execution time for the web page with only one ScriptSpace. The first count for the second handler was recorded 251 msec after the first handler had completed execution.](image)

We observed that the execution never overlapped for the first page. The second click handler started executing only after the previous handler had completed execution. This serialization occurred because the thread processing the click event for the first button obtained the ownership of the ScriptSpace while the second event handling thread waited for the first
Figure 5.2: Progress of scripts associated with separate ScriptSpaces. The plot was obtained by recording the number of counts to 10000 vs. the execution time when click handlers were associated with different ScriptSpaces. The execution of the handlers overlapped.

thread to release ownership, thus maintaining the single threaded execution environment for the scripts.

For the second web page, the onclick handlers of both buttons were associated with different ScriptSpaces. Therefore both of the event handling threads were able to obtain the ownership of their ScriptSpaces right away. We observe from graph 5.2 that the execution of the handlers proceeded in parallel.
CPU Resource Control

Our implementation provides fair sharing of CPU time when different ScriptSpaces contend for CPU time. In this experiment, we evaluate the ability to control allocation of CPU time to ScriptSpaces with different CPU shares. We also examine if we can change the CPU allocation by changing the CPU shares of the ScriptSpaces according to a resource control policy.

We simulated a CPU bound load by running three scripts in a page that repeatedly incremented an integer from 0 to 100 in each iteration and displayed the value in the document. The counting scripts were registered as click handlers of three buttons, placed inside separate div containers with different ScriptSpaces identified by the ScriptSpace attributes \texttt{SS\_1}, \texttt{SS\_2}, and \texttt{SS\_3}. We began execution of the handlers by clicking on the buttons and recorded the number of iterations completed by each ScriptSpace over time as a measure of CPU time received. A polling script retrieved this information from the \texttt{nsIScriptSpaceRuntime} instance every other second. We divided the experiment into three stages, varying the weights of the ScriptSpaces over time. We obtained graph \ref{fig:count} by plotting the counts per second measured over the execution time for each ScriptSpace.

Stage\#1

In the first stage, the weights associated with ScriptSpaces \texttt{SS\_1}, \texttt{SS\_2}, and \texttt{SS\_3} were 30, 1, and 50 respectively. \texttt{SS\_1} and \texttt{SS\_3} started counting with an average of 34 and 57 iterations per second, sharing 37\% and 61\% of the CPU time according to their weights and total weight ratio. Since \texttt{SS\_2} was assigned the minimum weight, it received much less CPU than the other handlers and its progress was negligible. We observed occasional red dots for \texttt{SS\_2} in \ref{fig:count} illustrating its slow progress.

Stage\#2

At a later point, we changed the weight of \texttt{SS\_2} to 50 and observed the changes in the work done by the ScriptSpaces. The number of iterations completed by \texttt{SS\_2} per second spiked up. This also lead to a dip in the CPU shares of each of \texttt{SS\_1} and \texttt{SS\_3}. During this time the total weight changed from 81 to 130 and \texttt{SS\_1}, \texttt{SS\_2}, and \texttt{SS\_3} started receiving 24\%, 38\%, and 38\% of the CPU. A similar change was observed in the work done. The occasional red
Figure 5.3: Illustration of CPU resource control. The area chart is obtained from the number of iterations completed by the event handlers per second during the experiment. Initially the event handler associated with ScriptSpace $SS_2$ could not make much progress as its CPU share was low. When the CPU share was increased for $SS_2$, it received more CPU time and its execution sped up. However, CPU allocation for the other two executions reduced. Finally, the CPU share for $SS_2$ was reset to its original value to prevent $SS_2$ from obtaining CPU time.

The color of $SS_2$ representing one iteration changed to long red stripes with an average iterations of 32.

**Stage #3**

We assumed a policy that considered the progress of $SS_1$ and $SS_3$ to take precedence over $SS_2$. Therefore, we reduced the weight of $SS_2$ to the minimum weight of one to prevent
SS₂ from “stealing” CPU allocations from SS₁ and SS₃. This led to an increase of the CPU time allocated to the other two ScriptSpaces since the total weight was reduced. We observed in the final portion of plot 5.3 that the counting speed of scripts associated with SS₁ and SS₃ increased again to the intended values. The progress of the execution associated with SS₂ was minimal from that point on.

![Service Error Graph](image)

**Figure 5.4**: Service error graph. The service error plot represents the difference in the number of iterations actually completed per second and the expected iterations per second for each ScriptSpace.

**Service Error**

We also measured the service error during the experiment. The results are presented in graph 5.4. The service error was calculated by subtracting the actual iterations per second from the
expected value based on the weight and total weight ratio for all of the ScriptSpaces. During
the time frame of the experiment, we observed that the average service error was bounded
by $-5$ and $+5$ iterations which was within 6% of the total iterations done per second, 86,
by all the ScriptSpaces. This low service error indicated that our scheduler implementation
was successful in sharing the CPU time among the executions proportional to the weights
of the associated ScriptSpaces.

5.2 Robustness Goals

Thwarting DOS Attacks

In addition to DOM event handlers, the browser executes scripts which are network callbacks
for XHR and timer expiration handlers embedded in a page loaded in the browser. In this
experiment, we show that the resource control ability in a ScriptSpace enabled browser is
comprehensive by subjecting it to denial-of-service attacks initiated from these different types
of scripts.

The experiments were performed on both the unmodified Firefox codebase and the
ScriptSpace enhanced codebase. In all of the following experiments, we monitored the ex-
ecution of a timeout handler while simulating denial-of-service attacks aimed at the CPU
by running infinitely looping scripts. The timeout handler in the experiment was triggered
every 300 ms interval, did some calculation and returned. We have used the execution of
the timeout handler as the measure of resources allocated by the scheduler to a well-behaved
application while such attacks were in progress.

We used the following pages to launch the infinite loops:

- Page 1 containing a button with a click handler that triggered an XHR request to the
  server that hosted the page. The onreadystatechange handler for the request started
  an infinite loop.

- Page 2 containing a button with a click handler that started an infinite loop.

- Page 3 containing a button with a click handler that registered a timeout handler which
  executed an infinite loop.
We programmatically simulated click events on the buttons of each page so that the infinite loops were executed.

![Figure 5.5: Graph for DOS attacks in vanilla Firefox. In vanilla Firefox, the infinite loops successfully hogged the CPU, so that the timeout handler could not execute while the loops were in progress. The browser displayed a dialog asking the user’s consent to stop the script at some point. The timeout handler resumed execution only when the CPU hogging script was stopped.](image)

We plotted the number of timeout invocations against the execution time for each of the browsers and obtained graph 5.5 for the vanilla Firefox and graph 5.6 for the modified Firefox. We observed from the graph 5.5 that the invocation of the timer remained steady until the XHR onreadystatechange handler was fired. The infinite loop blocked the only thread of script execution in vanilla Firefox and caused the timer invocation to starve. The browser lost responsiveness as none of the buttons and menus in the browser could be clicked. After a few seconds, the browser displayed the busy script dialog box asking the user’s consent.
Figure 5.6: Graph for DOS attacks in modified Firefox. In our implementation, the infinite loops are isolated from the timeout handler execution as each one of them was associated with separate ScriptSpaces that were created for each page. The scheduler shared the CPU time among the CPU bound ScriptSpaces, while providing a scheduling boost to each timeout invocation.

to stop the script execution. We stopped the execution so that the timeout handler could resume execution. We marked this period with the red vertical lines for XHR in Figure 5.6. A similar observation was made when the CPU bound DOM event handler and the CPU bound timer expiration handler were executed. We observed that the Vanilla Firefox was not able to provide CPU time to the timeout handler while the infinitely looping scripts were hogging the CPU.

On the other hand, the timeout handler was executed on time in our implementation as observed from the graph Figure 5.6. All of the infinite loops were associated with the default
ScriptSpace of the pages they belonged to. At the beginning, the scheduler considered them to be a new event and warped their execution. That caused a small dint in the graph at the time each of these CPU bound scripts started. We marked them with vertical yellow lines in the graph. After the boost subsided, the scheduler started sharing the CPU time among those ScriptSpaces according to their weights. The timeout handler, which was fired every 300 msec belonged to the default ScriptSpace of that page. It always received a warp and was executed immediately to completion. The browser remained responsive to user interaction during the experiment. We observed that the ScriptSpace enhanced browser was, thus, able to control CPU allocation to all ScriptSpaces in the browser and thwart the DOS attacks launched by different types of scripts.

Isolation within a Mashup

Mashups integrate third-party code and data in the same page. In this experiment, we show that our implementation can provide isolation to real mashups with almost no changes to the code.

We created two simple applications in JavaScript called *Fibonacci Calculator* and *Timer* and added them to the iGoogle home page of a user as listed in Appendix A. The Fibonacci Calculator calculates the Fibonacci value for a given number using the recursive Fibonacci Progression $^1$. The time complexity of calculating the Fibonacci value for a number $n$ is $O(2^n)$ using the recursive algorithm. This gadget represents a CPU bound execution. On the other hand, the timer gadget represents a latency sensitive application. The timer is fired every 500 msecs. In each timeout, the current time is calculated and displayed on the gadget.

For the experiment, we loaded the iGoogle homepage for a test user in the vanilla Firefox as well as in the modified Firefox and recorded the timeout events for the timer gadgets while calculating the Fibonacci value for the number 30. We obtained graph 5.7 after plotting the number of timeout invocations against the execution time for the vanilla Firefox.

Since the unmodified Firefox executes scripts within a single thread, the CPU bound Fibonacci calculation blocked the thread. The timeouts were not executed and the current time was not updated on the web page until the calculation completed, as observed from the

---

$^1$The Fibonacci Progression is a numeric progression defined as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. 
horizontal portion of the graph between the 19th and 20th invocations.

Figure 5.7: Execution of scripts in iGoogle gadgets in vanilla Firefox. The CPU bound Fibonacci calculation prevented the timer gadget to update the current time on the page in original Firefox. Even though the two gadgets were separate applications loaded in the iGoogle mashup, the browser could not isolate them.

In modified Firefox, default ScriptSpaces were created for both the gadgets as they were embedded in the iGoogle page in different iframes. Therefore, both of the scripts were able to execute independently of each other. That allowed the periodic timer invocations to proceed with no delays as seen from the graph.
Figure 5.8: Execution of scripts in iGoogle gadgets in modified Firefox. Each gadget is embedded in separate iframe elements in iGoogle. Our implementation created default ScriptSpaces for each of the gadgets. Isolated in its own ScriptSpace, the timer gadget was able to execute the timeout handler and update the current time on the web page, thus maintaining a useful service for the user.
Chapter 6

Related Work

We discuss research activities that isolate web content using the protection facilities provided by OS processes in section 6.1. In section 6.2, we discuss concurrent research work that isolates web content for web security.

6.1 Content Isolation using OS Processes

Tamoma [CGLH06] is an experimental virtual machine environment developed for isolating browser instances in a host operating system. In Tamoma, an instance of a browser runs sandboxed on top of a browser operating system or BOS layer. BOS instantiates and manages the sandboxes by managing accesses to system resources. The web applications loaded in the browser are isolated from local resources within each sandbox, removing the need to trust web browsers and the web applications.

The OP browser [GSK08] improves the granularity of isolation by dividing the core components of a browser into five subsystems: network, UI component, web page subsystem, storage component, and a browser kernel. Each subsystem runs as a separate OS process. The browser kernel component manages the subsystems, manages the communications among them, and communicates with the underlying operating system. The web page subsystem creates separate processes to load the HTML component and the JavaScript interpreter for each web page. Plug-ins in the web pages are instantiated as separate processes as well. Separating each of the browser components in separate processes enables the OP browser to isolate any failure within a page or a failed browser plug-in from the rest of the browser.
Above research activities have influenced the process-per-tab isolation approach found in browsers like Google Chrome and IE8. However, as we have shown, isolation of a web page as a process cannot provide isolation to components in mashups. On the contrary, ScriptSpaces can provide isolation for script execution across pages as well as within a mashup as it provides finer grained isolation.

MashupOS \cite{WFHJ07} is an experimental browser-based operating system for isolating content that belongs to different domains by supporting separate security domains in a web page. It provides an abstraction called ServiceInstance for protection and communication among different applications loaded from different domains in mashups. Orthogonal to that, ScriptSpaces isolate script content within a web page to provide resource control during execution of scripts.

## 6.2 Content Isolation for JavaScript Security

Several research activities have changed the way scripts are embedded in web pages in order to ensure safety from malicious script content. For example, Caja \cite{MSL08} is an object capability based solution for ensuring safety when scripts from third-party web sites are embedded in a page. An object capability model ensures that an object can invoke only the public interfaces of the objects to which it holds references. A source-to-source translator translates the user script code to a subset of JavaScript, removing troubled language constructs like `eval` and `with` from the code and grouping the code into *Caja modules*. This ensures that access to browser global objects such as the `window` object is mediated, and any unwanted modification to such objects is denied.

In a similar approach, ADsafe \cite{Cro10a} tries to integrate third-party content, such as an advertisement into a web page by safely adding the content as part of a customized JavaScript object called `ADSAFE` so that accesses to global objects and DOM can be mediated. Like Caja, ADsafe disables the direct use of certain unsafe language constructs in the JavaScript language and provides its own safe interfaces to them. The content writers can use tools like JSLint \cite{Cro10b} to verify that their code conforms to ADsafe guidelines.

ConScript \cite{ML10} is another approach that tries to constrain the functionality provided by JavaScript using a client side advice implementation for security. ConScript modifies the existing JavaScript runtime in IE8 to support an aspect system \cite{Kit96} that enables a web
page to control the script invocation at runtime. ConScript defines an *around advice* for scripts that takes a function call that needs to be advised as the first parameter, and a corresponding function that needs to be executed instead of the advised function as the second parameter. It modifies the JavaScript engine to recognize and execute the advice when a script execution makes a call to the advised function.

All the above approaches restructure script code into separate groups and provide a separate namespace to each such group. This separation prevents a script from making unintended modifications outside its namespace. However, such approaches do not consider resource control. Such solutions could coexist with ScriptSpaces to provide an extra layer of robustness.
Chapter 7

Future Work and Conclusions

Before we summarize our contributions and conclude, we discuss some of the directions in which our work can be continued.

7.1 Future Work

Our current implementation of ScriptSpaces does not provide memory resource control. In order to prevent denial-of-service attacks aimed at memory resources, the browser must be able to control the memory allocation and deallocation of scripts. Memory management for ScriptSpaces can be split in three parts:

1. Isolating memory allocation per ScriptSpace: Each script should be able to allocate all its objects on a fixed heap maintained per ScriptSpace. A separate heap per ScriptSpace can be used to provide protection for memory accesses by preventing arbitrary access to the heap by scripts associated with other ScriptSpaces.

2. Garbage collection per ScriptSpace: In our present implementation, we rely on a single, reference counting garbage collector in Firefox that loops over all JavaScript contexts active in the system to collect unreferenced memory. The use of one garbage collector per ScriptSpace would allow charging the CPU time consumed during garbage collection to individual ScriptSpaces.

3. Handling Out of Memory condition safely: A ScriptSpace may run out of memory while executing a script. We would like to detect such situations, and handle such errors
safely so that the integrity of other ScriptSpaces or the browser is not compromised.

At present, we have not evaluated the performance of our implementation against available browser implementations as we have considered performance to be orthogonal to the isolation of web content. However, the codebase of Firefox maintained by Mozilla has undergone several performance enhancements since we have started our implementation. In future, we plan to port our changes to the latest version of the Firefox codebase in order to facilitate a realistic evaluation of the cost and benefits provided by ScriptSpaces.

7.2 Conclusions

We presented an abstraction called ScriptSpace to support the isolation of scripts in web browsers in this thesis.

Existing web browsers consider a web page as the entity for isolation. However, many web pages, including mashups, embed code and data from multiple web sites within a single page, making isolation of web content within a page necessary. ScriptSpaces provide separate, isolated execution environments for scripts within HTML containers in a web page as well as across web pages loaded in the browser.

We have shown that the isolation within a page enables concurrent script execution within the page. Misbehaving scripts within a mashup can be identified and isolated.

A ScriptSpace also acts as the resource principal for a script execution. Our implementation currently provides CPU resource control. A version of BVT scheduler is used in our implementation to share the CPU among multiple ScriptSpaces. We provide a ScriptSpace API for the management of CPU resources from both C/C++ and JavaScript code. We developed a utility on top of the API called SSManager that can be used to monitor and control CPU consumption by scripts at the ScriptSpace level. We demonstrated that our prototype successfully isolates misbehaving scripts in mashups like iGoogle. We found that the ScriptSpace enabled browser exhibited a low average service error during CPU bound script executions.
Bibliography


Appendix A

iGoogle Gadgets

The Fibonacci Calculator Gadget

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="Fibonacci Calculator!"/>
  <Require feature="opensocial-0.8" />
</ModulePrefs>
<Content type="html">
<![CDATA[
<script type="text/javascript">
function invokeFibo()
{
  var num = document.getElementById("my_number");
  var timeStamp = new Date();
  dump("fibo:start:"+num.value+":"+timeStamp.getTime()+"\n");
  var n = fibo(num.value);
  timeStamp = new Date();
  dump("fibo:end:"+num.value+":"+timeStamp.getTime()+"\n");
  var n = fibo(num.value);
  var result = document.getElementById("my_value");
  result.innerHTML = n;
}
function fibo(n)
{
  if(n <= 0){
    return 1;
  }else{
    return fibo(n-1)+fibo(n-2);
  }
}
</script>
<div>
Enter a number : <input id="my_number" type="text" />
<input type="button" value="Calculate Fibonacci" onclick="invokeFibo()"/>
</div>
```
The Timer Gadget

Listing 1: Fibonacci Calculator

```xml
ten
<!DOCTYPE html>

<ModulePrefs title="Timer G!">
  <Require feature="opensocial-0.8"/>
</ModulePrefs>

<Content type="html">
  <![CDATA[

    <script type="text/javascript">
      function startTime() {
        var currTime = new Date();
        var h = currTime.getHours();
        var m = currTime.getMinutes() < 10 ? "0" + currTime.getMinutes() : currTime.getMinutes();
        var s = currTime.getSeconds() < 10 ? "0" + currTime.getSeconds() : currTime.getSeconds();
        document.getElementById('my_value').innerHTML = h + ':' + m + ':' + s;
        window.setTimeout(startTime, 500);
        dump(currTime.getTime() + "\n");
      }
      gadgets.util.registerOnLoadHandler(startTime);
    </script>

    <div>
      Current Time: <h3 id="my_value"></h3>
    </div>
  ]]>
</Content>
</Module>
```

Listing 2: Timer Gadget