Splitter: A Proxy-based Approach for Post-Migration Testing of Web Applications

Xiaoning Ding[1], Hai Huang[2], Yaoping Ruan[2], Anees Shaikh[2], Brian Peterson[3], Xiaodong Zhang[1]

[1]Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210 USA
{dingxn, zhang}@cse.ohio-state.edu

[2] IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
{haih, yaoping.ruan, aashaikh}@us.ibm.com

[3]IBM CIO Office
Somers, NY 10589 USA
blpeters@us.ibm.com

Abstract

The benefits of virtualized IT environments, such as compute clouds, have drawn interested enterprises to migrate their applications onto new platforms to gain the advantages of reduced hardware and energy costs, increased flexibility and deployment speed, and reduced management complexity. However, the process of migrating a complex application takes a considerable amount of effort, particularly when performing post-migration testing to verify that the application still functions correctly in the target environment. The traditional approach of test case generation and execution can take weeks and synthetic test cases may not adequately reflect actual application usage.

In this paper, we propose and evaluate a black-box approach for post-migration testing of Web applications without manually creating test cases. A Web proxy is put in front of the production application to intercept all requests from real users, and these requests are simultaneously sent to the production and migrated applications. Results generated by both applications are then compared, and mismatches due to migration problems can be easily detected and presented to testing teams for resolution. We implement this approach in *Splitter*, a software module that is deployed as a reverse Web proxy. Through our evaluation using a number of real applications, we show that *Splitter* can effectively automate post-migration testing while also reduce the number of mismatches that must be manually inspected. Equally important, it imposes a relatively small performance overhead on the production environment.

Categories and Subject Descriptors C.4.0 [*Performance of Systems*]: Reliability, availability, and serviceability; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

General Terms Design, Experimentation, Management, Verification

1. Introduction

The maturity of virtualization technology, along with the emergence of cloud computing service offerings [Hayes 2008], has driven a renewed interest in enterprises to migrate their IT operations to new (more capable) platforms. These new virtualized environments strengthen the value proposition of traditional server consolidation initiatives by offering improved flexibility and faster deployment, reduced energy costs, and potentially lower management costs.

While the end-state of such transformation efforts promises many benefits, the cost and effort required to migrate enterprise applications from their original legacy environments onto new target platforms (or into the cloud), can be daunting. The migration process is complex and labor-intensive, and typically requires a number of steps, including: collecting information about the source operating environment, readying the target environment, performing the migration, testing the application on the target environment, and finally, cutting over to the new environment. Several of these steps involve tasks that can be automated using existing management tools, such as application and system configuration discovery [BMC, IBM a], provisioning planner [IBM b], software patching, and sometimes even the migration itself (e.g., using P2V transformation tools [Novell]).

Relatively little automation or tooling exists, however, to automate post-migration testing which is crucial to ensure that an application functions and performs adequately in the target environment. In our experience, migration is an error-prone process. Even though code change is usually not needed for migration, many applications often require re-

EuroSys'10, April 13–16, 2010, Paris, France.

configuration or tweaking after they are migrated to accommodate changes in the target environment such as changes in middleware and library, operating system, and hardware setup. These configuration tweaks are often the common culprits to migration problems.

The current practice in post-migration verification of applications centers around functional verification testing (FVT), in which individual test cases are specially created for each migrated application. Clearly, FVT can be very time consuming to design and implement (in our experience, we have observed that some applications require up to 6 weeks in testing before they are ready for cut-over to production). For enterprise applications that have more complex interactions with other applications or services, more time is often needed. Although there have been some efforts to reduce testing time by automating test generation and execution (e.g., [Benedikt 2002, Elbaum 2003, Lucca 2002, Ricca 2001]), there is still a significant level of human involvement when preparing inputs for test cases and examining outputs. Moreover, existing approaches usually generate test cases by modeling and analyzing applications, e.g., at the source code level, and thus, the generated test cases might not reflect how real users interact with applications.

To reduce the human efforts required in testing, we propose a new testing approach that uses actual user activities to test migrated applications against their production counterparts. We treat both production and migrated applications as black boxes, and use identical workloads to exercise them simultaneously during the testing phase. The response from the migrated application is compared to the production application's response, and if they are identical, the migrated application is assumed to be working correctly. Sometimes, even when responses are not identical, the migrated application may still be operating correctly due to normal variances that exist among different application instances. As a result, we also developed heuristics to properly handle these discrepancies.

Our primary focus in this paper is on testing Web-based applications, and proposed techniques are specifically tuned to HTTP protocol. Our methodology, although, can be used to test other types of applications, middlewares, and systems, the actual implementation will be significantly different due to protocol differences, e.g., replicating a HTML request with all the correct session information is very different from replicating a SQL command.

We believe the benefits of our approach are three-fold. First, and most significantly, there is no need to design or generate test cases, which saves considerable time and effort. Second, because the migrated application is tested with real user workloads rather than synthetic test cases, the results are more likely to represent how applications will be exercised in a production environment. Finally, since we perform the testing of the migrated application while the production application is still active and authoritative, there is less manual analysis needed to determine whether a response is in fact correct or not.

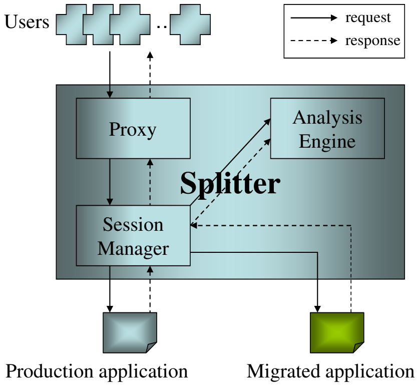


Figure 1. Splitter system architecture.

We implement this testing approach in a tool called *Splitter*. Using five Web-based applications, we evaluate *Splitter* in terms of i) its ability to correctly flag different types of migration problems, ii) its performance impact on the production application, and iii) its ability to identify false positives. Based on our fault injection experiments, we found that *Splitter* is able to successfully detect different classes of migration errors that are often encountered in practice. *Splitter* only introduces a relatively small impact on throughput (less than 5% in our tests) in the request forwarding path, which should be sufficiently small for deployment in all but the most delay-sensitive production environments.

In the next section, we give an overview of the design and implementation of *Splitter*. *Splitter* is put to test on several applications and workloads, and the results are given in Section 3. This is followed by a discussion section (Section 4), where we discuss our deployment experience of the tool and some of its limitations. Related work is discussed in Section 5, and finally, we summarize and conclude in Section 6.

2. Design and Architecture

Splitter tests migrated applications by observing their external behaviors, i.e., their responses for HTTP requests. This is not only because the external behaviors of an application can reflect its functionalities, but also because the external interface is usually less complicated than the internal running environment that involves various hardware and software components. As we will show later, Splitter also leverages the well-defined syntax and semantics of HTTP requests and responses to reduce testing complexity and to improve testing quality.

To observe the external behaviors of Web applications, Splitter provides 3 capabilities: i) capture and replicate user requests, ii) instrument the replicated requests so they are meaningful to the migrated application, and iii) identify migration problems from inspecting application responses. Figure 1 shows the overall architecture of Splitter. It consists of three components: a *Proxy* that sits in front of the production application, a *Session Manager* that manages various aspects of instrumenting HTTP requests, and an *Analysis Engine* that analyzes responses and provides a user interface to test engineers for identifying migration problems. Additional details of these components are described in the following sections.

2.1 Proxy

Splitter first needs to intercept HTTP user requests to replicate them. Since many Web proxies already provide this capability by interposing between users and application / Web servers, we simply added the request replication capability into a widely used Web proxy—Squid [Squid].

We call this component of Splitter a *Proxy*. This is the simplest of the 3 components. Due to it being placed on the critical path to the production application, it should only impose minimal impact to the application. Upon receiving a user request, Proxy immediately passes the request to the production application without any delay. Once the request is sent, it then makes a replica of the request and sends it to the Session Manager, which, in turn, will inspect the replicated request and make any necessary instrumentations before passing it to the migrated application.

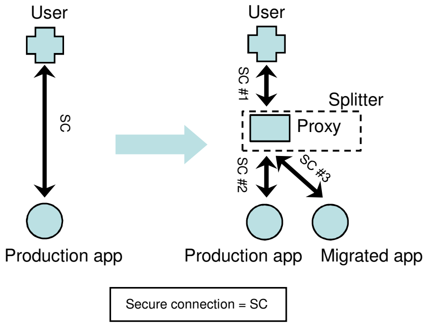
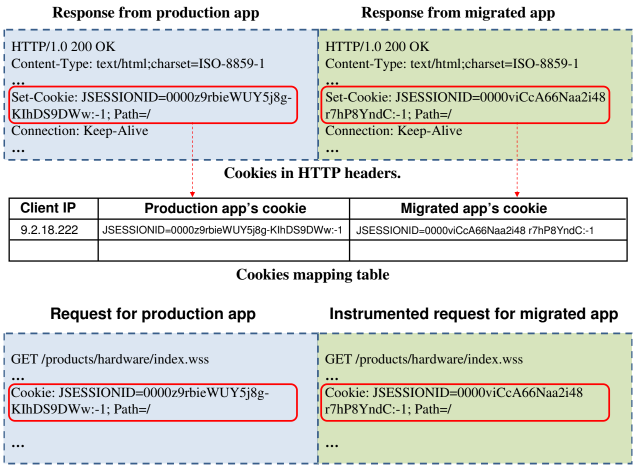


Figure 2. To allow visibility into encrypted payloads, we use three secure connections centralized at Proxy.

2.1.1 Encrypted Payload

SSL protocol is commonly used to support encrypted HTTPS traffic between users and Web applications to avoid eavesdropping of sensitive information, commonly found in business applications. This poses a problem for us as we need to "clearly" see all the traffic, not only so we can meaningfully compare returned responses but also for correctly replicating user requests. The key to solve this problem is to gain visibility into the request and response payload.

Our solution makes use of 3 separate secure connections: (i) between users and Proxy, (ii) between Proxy and the production application, and (iii) between Proxy and the migrated application, as illustrated in Figure 2. This allows Proxy to see all user traffic in clear text as we needed. However, to achieve complete user-transparency, appropriate SSL certificate needs to be installed on Proxy (otherwise, client browser will prompt users with a warning message).



Response from production app

```
HTTP/1.0 200 OK
Content-Type: text/html;charset=ISO-8859-1
...
Set-Cookie: JSESSIONID=0000z9rbieWUY5j8g-
KIhDS9DWw:-1; Path=/
Connection: Keep-Alive
...
```

Response from migrated app

```
HTTP/1.0 200 OK
Content-Type: text/html;charset=ISO-8859-1
...
Set-Cookie: JSESSIONID=0000viCcA66Naa2i48
r7hP8YndC:-1; Path=/
Connection: Keep-Alive
...
```

Cookies in HTTP headers.

Client IP	Production app's cookie	Migrated app's cookie
9.2.18.222	JSESSIONID=0000z9rbieWUY5j8g-KIhDS9DWw:-1	JSESSIONID=0000viCcA66Naa2i48 r7hP8YndC:-1

Cookies mapping table

Request for production app

```
GET /products/hardware/index.wss
...
Cookie: JSESSIONID=0000z9rbieWUY5j8g-
KIhDS9DWw:-1; Path=/
...
```

Instrumented request for migrated app

```
GET /products/hardware/index.wss
...
Cookie: JSESSIONID=0000viCcA66Naa2i48
r7hP8YndC:-1; Path=/
...
```

Figure 3. Building HTTP cookies mapping table and using it to instrument replicated requests.

2.1.2 Compressed Payload

Traffic between user browser and a Web server is sometimes compressed to reduce network load and to improve response time. Similar to encrypted traffic, comparing and instrumenting compressed payload are also not meaningful. Therefore, we de-compress and re-compress packets as needed.

The additional encryption/decryption and compression / decompression work inevitably adds more overheads. In practice, however, we were able to show that Proxy introduces only small impact to the production application, as we will show in Section 3.

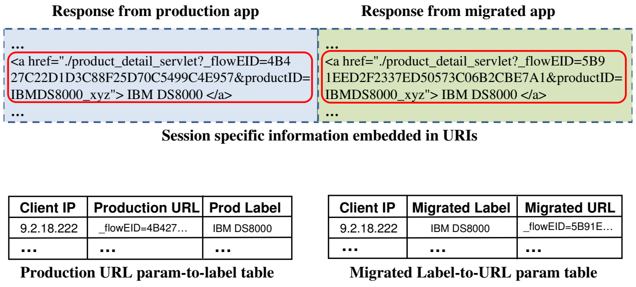
2.2 Session Manager

Directly forwarding replicated HTTP requests to the migrated application will often yield unexpected result as requests sometimes contain user-specific and/or server-specific state information. An example of this is HTTP cookie. In this section, we describe how we handle HTTP requests containing cookies, followed by other situations where HTTP requests require instrumentations.

2.2.1 HTTP Cookies

HTTP cookies are commonly used by Web applications to uniquely identify users during each session and/or across different sessions. They are generated by Web applications and are only recognizable by the application instance that generated them. For example, if we denote the production application as P and the migrated application as Q, and having Splitter forwarding a request containing a cookie C_p (generated by P) to Q, Q will not recognize the request and will generate an error.

This problem can be solved by substituting C_p with C_q (cookie generated by Q) before forwarding the request to Q. To do so, we keep a lookup table as shown in Figure 3. New


Response from production app
Response from migrated app
Session specific information embedded in URIs

```
...
<a href="./product_detail_servlet?_flowEID=4B4
27C22D1D3C88F25D70C5499C4E957&productID=
IBMDS8000_xyz"> IBM DS8000 </a>
...
```

```
...
<a href="./product_detail_servlet?_flowEID=5B9
1EED2F2337ED50573C06B2CBE7A1&productID=
IBMDS8000_xyz"> IBM DS8000 </a>
...
```

Client IP	Production URL	Prod Label
9.2.18.222	_flowEID=4B427…	IBM DS8000
…	…	…

Production URL param-to-label table

Client IP	Migrated Label	Migrated URL
9.2.18.222	IBM DS8000	_flowEID=5B91E…
…	…	…

Migrated Label-to-URL param table

Figure 4. Building URL parameters mapping table and using it to instrument replicated requests.

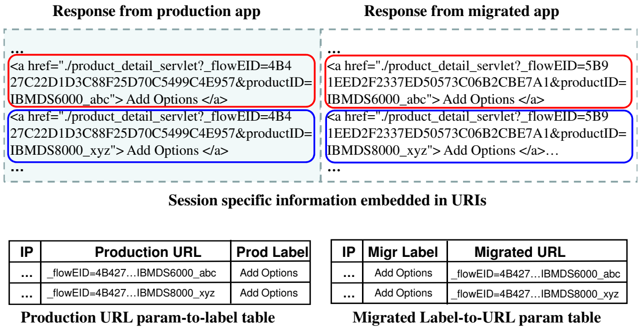
entries are entered into the table by inspecting *COOKIE:* and *SET-COOKIE:* fields in HTTP headers. This makes the table essentially a mapping from C_ps to C_qs. On subsequent user requests containing C_p, we find its corresponding C_q in the mapping table and replace it with C_q in the replicated user request before sending the request to Q. The mapping table is further indexed by clients' IP addresses since there might be simultaneous users.

2.2.2 URL Parameters

In addition to HTTP cookies, there are other types of elements for which instrumentation is needed. One such type of element is URL parameter. Web applications often include session identifying information within URL parameters, as shown in Figure 4.

Similar to cookies, some URL parameters like session IDs are also generated by Web applications and are specific to the application instance that generated it. We can use the same cookie substitution method to handle URL parameters, but the substitution is a bit trickier here. Instead of using only one mapping table for cookies, we need to use two mapping tables to handle the substitution. The first table is a mapping between URL parameters and labels (e.g., for HREF, label is the text between <A> and </A>) generated by the production application. The second table is a mapping between labels and URL parameters generated by the migrated application. An example is shown in Figure 4. In this example, if the user clicks on the link associated with the HREF tag, the browser will generate a GET request. After the GET request is replicated by Splitter, URL parameters of the request are looked up in the first table to find the label of the link. In this case, it is "IBM DS8000", which is used subsequently to look up in the second table to find the URL parameters we should substitute before sending the request to the migrated application.

The trickier part comes in when there are multiple labels that are identical as shown in Figure 5. This will break the substitution method described above. We briefly explored the idea of enumerating all the URL links with parameters in a page and using the count as the index to the mapping tables as opposed to Labels. However, we quickly found this would not work as links sometimes would get out of order due to some contents being dynamically generated at runtime.


Response from production app
Response from migrated app
Session specific information embedded in URIs

```
...
<a href="./product_detail_servlet?_flowEID=4B4
27C22D1D3C88F25D70C5499C4E957&productID=
IBMDS6000_abc"> Add Options </a>
<a href="./product_detail_servlet?_flowEID=4B4
27C22D1D3C88F25D70C5499C4E957&productID=
IBMDS8000_xyz"> Add Options </a>
...
```

```
...
<a href="./product_detail_servlet?_flowEID=5B9
1EED2F2337ED50573C06B2CBE7A1&productID=
IBMDS6000_abc"> Add Options </a>
<a href="./product_detail_servlet?_flowEID=5B9
1EED2F2337ED50573C06B2CBE7A1&productID=
IBMDS8000_xyz"> Add Options </a>…
...
```

IP	Production URL	Prod Label
…	_flowEID=4B427…IBMDS6000_abc	Add Options
…	_flowEID=4B427…IBMDS8000_xyz	Add Options

Production URL param-to-label table

IP	Migr Label	Migrated URL
…	Add Options	_flowEID=4B427…IBMDS6000_abc
…	Add Options	_flowEID=4B427…IBMDS8000_xyz

Migrated Label-to-URL param table

Figure 5. An example illustrating a potential problem with labels being the same.

Our solution is fairly simple. We do the first table lookup as described before. When we get the label from the first table, we proceed to do another lookup in the second table and find all entries with that particular label. Finally, we do a string comparison of the original URL parameters being requested to all the URL parameter list entries we found in the second table. The one that matches the closest to the original URL parameters is the one we use to perform the final substitution. In the example shown in Figure 5, if the user clicks on the DS6000 link, using the parameters in this link, we find the label "Add Options" label in the first mapping table. This label has two corresponding entries in the second mapping table. The session ID part (i.e., *_flowEID*) of the two entries is equally different from the original *_flowEID*. However, the first entry has the same *productID* value as the original request, and therefore, it is used in the substitution.

2.2.3 POST Parameters and Javascript

POST parameters embedded in HTML forms can also be substituted similarly as URL parameters since one can easily convert POST parameters to GET parameters and vice versa.

Javascript and AJAX are used very commonly nowadays to offload workloads from server side to client side. Their relevancy to this work is that they can construct a URL with a dynamic set of parameters, e.g., a Javascript builds a new URL by appending the sessionID to one or more other parameters to perform a particular action within the session. This list of parameters cannot be substituted using the mapping tables described above. To correctly substitute a dynamically generated list of parameters, we build a finer grained lookup table. For example, in Figure 5, we would build an entry for *_flowEID* and a separate entry for *productID*. When a new request comes in, we would first break down its list of parameters, and then consult this table to substitute each parameter one by one.

2.2.4 Request and Response Synchronization

The substitution method requires all the mapping tables to be correctly built. The entries in the mapping tables are built from parsing responses from production and migrated applications. We observed that to correctly make a substitution, some synchronization is required amongst the traffic streams of the production and migrated applications.

For example, assume in the i^{th} transaction, its request R_i (and its replica R'_i) contains parameters / cookies and they were generated in a previous transaction j, for which, we denote its response from the production application as Q_j, and the migrated application as Q'_j. To correctly substitute parameters in R'_i, we must have received both Q_j and Q'_j and populated the corresponding mapping tables. This means we need to queue up some replicated requests until all the necessary information are arrived. Since most requests have no parameters or cookies, this queuing only applies to a small percentage of requests. Furthermore, this does not impact the production application in any way since its request and response packets are simply handled as pass-thrus.

2.3 Analysis Engine

Analysis Engine compares and analyzes responses from the production and migrated applications to help test engineers to determine if the migration is successful. If a Web application only contains static data, the migrated instance should produce identical responses as the production application. However, most enterprise Web applications nowadays also support dynamic contents. The challenge in response comparison is to differentiate differences that are a result of migration errors from those that are a result of normal application variance, such as timestamps, advertisements, etc. To achieve this, we developed a content-based comparison algorithm and two heuristics to rank differences according to how likely they are caused by migration problems. These differences are further categorized and grouped for test engineers to analyze for finding root causes. The content-based comparison and categorization process is intended to be done offline due to the processing time required to parse complex Web application responses. However, we also provide a coarse-grained online screening mechanism to reduce both storage space required to collect request and response data and the offline processing time.

2.3.1 Online Screening

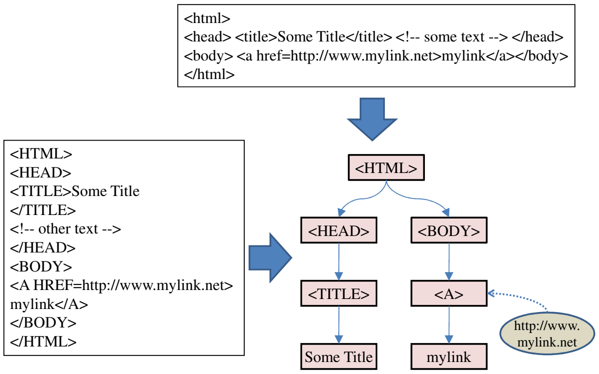
The online screening mechanism does a quick string comparison of HTTP headers and bodies at runtime. This is intended to throw away any data that is not useful to analyze afterwards, and thus, reduces storage requirements.

When comparing HTTP headers, we focus on three key fields, *Status Code, Content Type, and Content Length*, and we skip fields such as *Date, Cookies, Server, etc.* since difference of these fields do not imply incorrect behaviors of the application. This simple mechanism saves a significant amount of storage space as most of responses are identical. This mostly benefits large objects, such as images, documents, audio, and video files, which are often static and identical on both the production and migrated applications. These objects have been seen to account for over 85% of Internet Web traffic [Williams 2005]. We do not, however, screen and store any objects larger than a certain size (user customizable) due to both processing and storage overheads.

2.3.2 Finding Differences

The offline technique we use to compare responses depends on their MIME content types. For binary contents, we only need to know whether two binary objects are same or not. As the byte-to-byte comparison of binary objects has been carried out with online screening, binary contents are not future compared in this step. For text/plain and text/HTML contents, we do a more in-depth analysis as some differences are indications of migration problems and others just application normal variance.

- **Text:** for plain text response, Analysis Engine computes the *distance* between a pair of responses using Shorted Edit Script (SES) algorithm [Chawathe 1996]. *Distance* of two documents is defined as the number of characters to be inserted and/or deleted to change a document into the other. The *distance* value is divided by the total length of the HTTP body from the production application to yield a metric we called *Relative Distance*. We rank differences in Text data according to this metric—the higher the *relative distance* value, the more severe a problem is likely to be.



```
<html>
<head> <title>Some Title</title> <!-- some text --> </head>
<body> <a href=http://www.mylink.net>mylink</a></body>
</html>
```

```
<HTML>
<HEAD>
<TITLE>Some Title
</TITLE>
<!-- other text -->
</HEAD>
<BODY>
<A HREF=http://www.mylink.net>
mylink</A>
</BODY>
</HTML>
```

Figure 6. An example of comparing HTML contents with DOM trees

- **HTML:** SES is not very useful for comparing HTML pages since two HTML documents can be semantically the same at the HTML level even with many differences in the actual text, such as white spaces, comments, etc. Therefore, we convert HTML data to Document Object Model (DOM) [Wood 2000] tree format in which HTML tags are represented as nodes and their contents as leaves. An example of HTML DOM tree is illustrated in Figure 6. Analysis Engine compares two trees node-by-node in a breadth-first manner. Two nodes are said to be different if they have different tags or texts, or different number of child nodes. If two nodes are different, their sub-trees are not further compared.

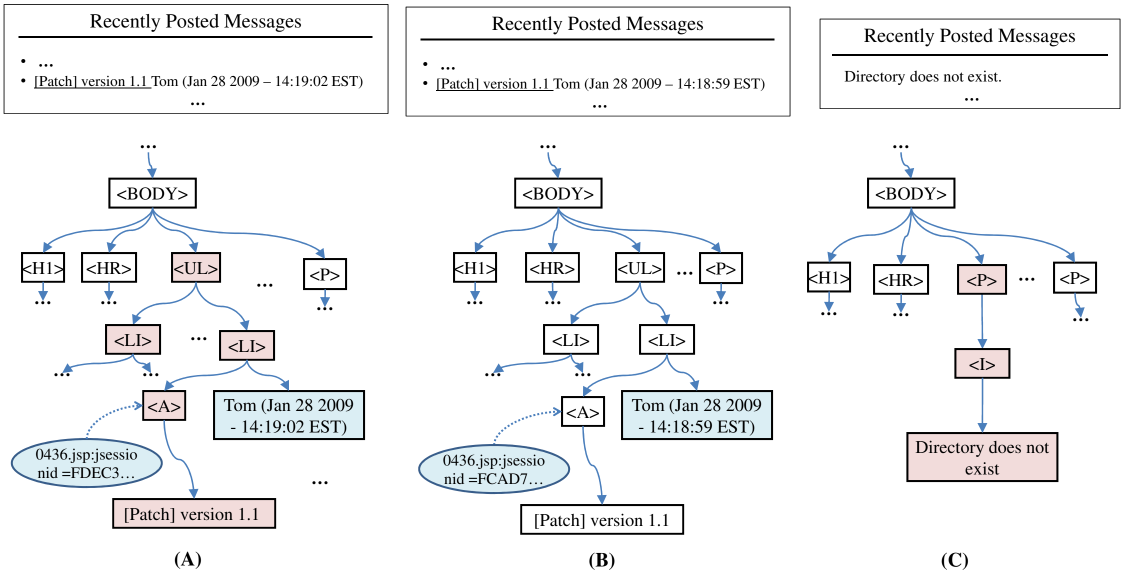

Recently Posted Messages
• …
• [Patch] version 1.1 Tom (Jan 28 2009 – 14:19:02 EST)
…
Recently Posted Messages
• …
• [Patch] version 1.1 Tom (Jan 28 2009 – 14:18:59 EST)
…
Recently Posted Messages
Directory does not exist.
…
<BODY>
<H1>
<HR>
<UL>
<P>
<LI>
<A>
Tom (Jan 28 2009 - 14:19:02 EST)
0436.jsp:jsessionid =FDEC3…
[Patch] version 1.1
Tom (Jan 28 2009 - 14:18:59 EST)
0436.jsp:jsessionid =FCAD7…
<I>
Directory does not exist
(A)
(B)
(C)

Figure 7. An example illustrating the structure heuristic. DOM trees are of the responses generated by the production application (**A**), a correctly-migrated application (**B**), and a problematic application on target system (**C**) for a same request.

2.3.3 Ranking Differences

To differentiate differences that are due to legitimate application variances from those that are caused by migration, we develop two heuristics to rank them, with higher ranked ones being more likely to be actual problems.

- **Structure heuristic:** when there are differences in the structure of the DOM trees in two HTML documents, it is almost always an indication of a real problem. As an example, Figure 7 shows responses of a real Web forum page. The responses are generated by i) the production application, ii) a correctly-running migrated application, and iii) a problematic migrated application for the same request. The first two responses have different *jsessionid* parameters embedded in their links (the attributes of <A> tags) and a different creation time associated with the links (shown on the right of the <A> tags). Both differences are legitimate runtime variations as the DOM tree structure is unchanged, indicating that the migrated application is probably working correctly. On the other hand, the response generated by the problematic migrated system has a completely different DOM tree structure, hinting a more likely problem. Of course, it is possible that a problematic application can generate a response with the same DOM tree structure as the production application and the only differences are textual differences within nodes. We identify such problems using another heuristic that we will describe shortly.

The relative importance of a particular difference is calculated as a function of its position within the DOM tree. The closer to the root node of the DOM tree, the more important a difference is since it affects all its child nodes in the tree. Specifically, the relative importance is calculated by counting the number of descendant nodes of the differing node and then dividing it by the total number of nodes in the DOM tree. This calculation is performed on the DOM tree generated by the production application.

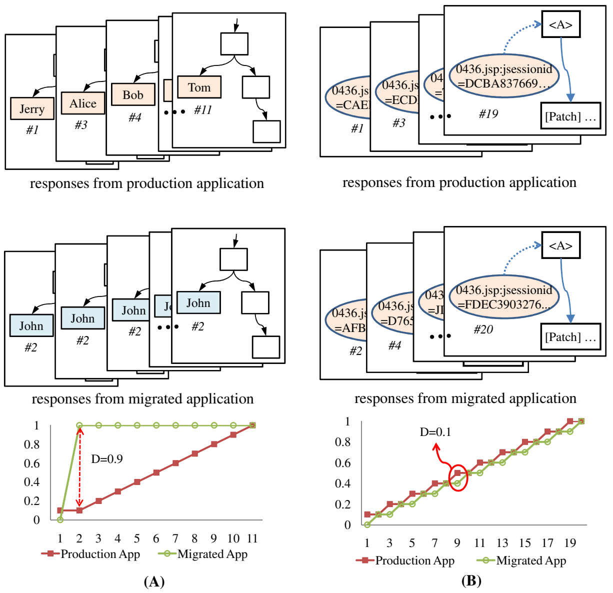


Figure 8. Two examples illustrating the distribution heuristic. Differing data are in shadow boxes, and are indexed. KS-test shows that differing data in example in sub-figure(A) may follows different distribution and differing data in example in sub-figure (B) may not.

- **Distribution heuristic:** if the only differences between a response pair are on the leaf nodes of the DOM trees (as opposed to structural differences), we cannot conclude the response is erroneous because it might be just normal variance such as timestamps. *Distribution heuristic* examines the value distribution of leaf nodes for similar requests of the entire testing period. If the distribution of the data in the responses from the migrated application is different from that of the production application, this indicates that the target system may not be working correctly. Figure 8 illustrates the distribution heuristic with two examples. In example A, data from the production application always changes, from “Jerry” to “Alice”, to “Bob”, et. al, while corresponding data from the migrated application only shows “John”. The invariance of the migrated application suggests that it may not be performing identically as the production application. In example B, the “jsessionid” fields in the responses from both applications change frequently, indicating that they may follow similar distributions. Changes of such a data field may be caused by normal runtime variations, thus they are less interesting for test engineers to examine.

We use two-sample Kolmogorov-Smirnov statistical test (KS-test) [Young 1977] to quantify the differences. KS-test computes a distance, called D statistic, between the cumulative distribution functions (CDFs) of the values in two datasets, and compares to a critical value $D_\alpha(m,n)$ for a given significance level α and the datasets of sizes m and n. If the D statistic is greater than $D_\alpha(m,n)$, the two datasets are considered to have different distributions.

To apply KS-test, Analysis Engine maintains all of the distinct values of the data field, and assigns an index to each of the values. Then it calculates CDF of the indexes, computes the D statistics, and compares it with the corresponding D_α. In our experiments, we set α to 0.1. In example A, the D statistic is 0.9, which is greater than $D_{0.1}(10,10) = 0.6$, suggesting that the response data follows a different distribution. In example B, the D statistic is 0.1, and is less than $D_{0.1}(10,10)$, implying that the two data sets have very similar distributions.

2.3.4 Categorizing Differences

We observe that requests are highly repetitive in Web applications, leading to redundant alerts. Moreover, the number of erroneous responses could be enormous if an application is not correctly migrated. Even with the help of Analysis Engine, it is still very time-consuming for test engineers to review all the errors. We try to alleviate this problem by grouping related problems into smaller number of categories using the following steps. (1) group together all responses triggered by the same request URI. (2) For URLs with parameters, we put those having the same keys (but may be with different values) into one category. For example, responses for *http://foo.com/service?category=business&id =BUS074* are classified into the same group with responses for *http://foo.com /service?category=business&id=BUS661*. However, responses for *http://foo.com/service?category= education&state=NY* would be in a different category because it has a different parameter key *state*. For some special applications, key values also matter if they can affect the

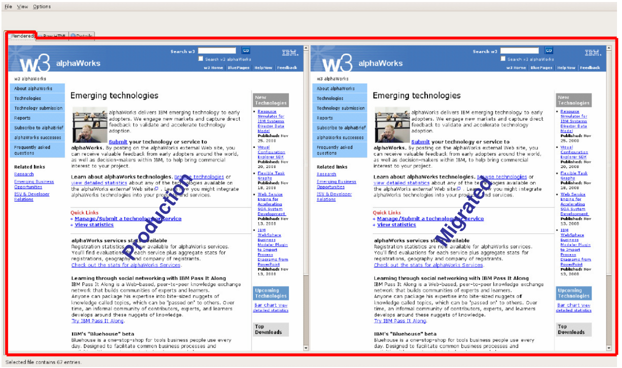
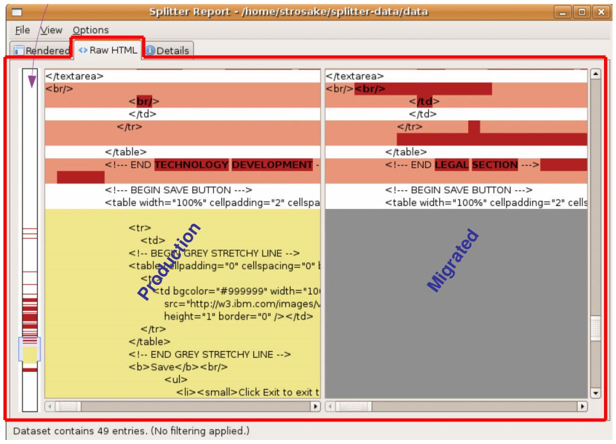


Figure 9. Analysis Engine’s browser rendered view.



```
</textarea>
<br/>
        <br/>
        </td>
      </tr>

     </table>
     <!--- END TECHNOLOGY DEVELOPMENT -

     <!--- BEGIN SAVE BUTTON --->
     <table width="100%" cellpadding="2" cellspa

        <tr>
          <td>
        <!-- BEGIN GREY STRETCHY LINE -->
        <table cellpadding="0" cellspacing="0" b
          <t
            <td bgcolor="#999999" width="10
              src="http://w3.ibm.com/images/v
              height="1" border="0" /></td>
          </tr>
        </table>
        <!-- END GREY STRETCHY LINE -->
        <b>Save</b><br/>
              <ul>
                <li><small>Click Exit to exit t
```

```
</textarea>
<br/><br/>
        </td>
        </td>
      </tr>

     </table>
     <!--- END LEGAL SECTION --->

     <!--- BEGIN SAVE BUTTON --->
     <table width="100%" cellpadding="2" cells
```

Dataset contains 49 entries. (No filtering applied.)

Figure 10. Analysis Engine’s raw HTML view.

overall structure of response pages. For example, responses for *http://foo.com/service?action=NewOrder* and responses for *http://foo.com/service?action=CancelOrder* can be very different, and thus should be put into different categories. Fortunately, such keys can be easily identified as they usually have much smaller cardinalities than other keys (e.g. customer ID and item ID). (3) For the responses in the same group, we further categorize them by the positions where differences occur, e.g. similar HTTP header fields or DOM tree nodes. (4) We rank the responses in each group according to their *relative distances* or *relative importance*.

2.3.5 Analysis Engine User Interface

To help test engineers interpret test data, in addition to console outputs presenting the detailed differences in responses, we also implemented a user interface to present these analysis results in an intuitive way. Test data can be viewed by test engineers in multiple perspectives. First, we present the data in a *Rendered View* as if the data was viewed by real users. A screen-shot of this view for a real application is shown in Figure 9. This view allows test engineers to get an at-a-glance view of a side-by-side comparison of the response data from the production (left) and migrated (right) applications.

When this view is insufficient to identify what exactly the differences are (some differences might not be apparent at the rendered view, e.g., Javascript), we provide a *Raw HTML View* to get the next level of details. A screen-shot of this view is shown in Figure 10. This is also a side-to-side comparison, but at the raw HTML level. We use various color coding to indicate different types of differences at the HTML code level to help test engineers to pinpoint migration problems.

3. Evaluation

In this section, we evaluate Splitter in terms of correctness, effectiveness and overheads by using a set of real Web applications and workloads. We first examine false positives, then describe experiences by injecting faults, finally evaluate performance overheads.

3.1 Experimental Setup

We used three machines in our experiments, all within the same LAN. Two Dell Optiplex 620 desktops (3GHz P4 CPU and 3GB memory) are used as production and migrated systems. Splitter is deployed on a Dell Dimension 3100 desktop (3GHz P4 CPU and 1GB memory). All systems are installed with RedHat Enterprise Linux Workstation 4.0. We use Apache 2.0.52 as Web server and Tomcat 5.5.27 as application server.

We select 5 Web applications in our experiments, each representing a different type of Web application. *TPC-W* [Marden 2001] is a transactional Web benchmark that simulates the activities of a business oriented transactional Web server. *Bookstore* is a fully functional online store application and represents a typical e-commerce application. *Portal* is an online Web portal application. *Forum* is an online discussion board that supports features such as interactive discussion threads, keyword search, and column sorting. *Yellow Page* is a Web-based yellow page system that supports information listing, searching, and administration of listings and users.

For *TPC-W*, we use its client software RBE (Remote Browser Emulator) to generate workloads. For other applications, we use DejaClick [AlertSite] to record multiple user sessions and replay them as our workload. DejaClick is a plug-in for Firefox.

3.2 False Positives

We first evaluate Splitter's ability to correctly replicate requests and to handle normal application variances when an application is correctly migrated. Two instances of each application are identically setup on both the production and migrated systems and their databases populated with the same data. We expect that either responses are the same and Analysis Engine would not flag any transactions, or responses are different but Analysis Engine is able to suppress them sufficiently so they are not presented to test engineers.

The comparison confirms that most responses are identical. However, Analysis Engine still detects some minor differences. In TPC-W, the differences are in the promotion banner where randomly selected books are recommended to clients and in the jsessionid parameter string embedded in URI links. In other applications, these differences include advertisements displayed in ads banners and timestamps of some events, e.g. the time when an order was placed in an online store, the time when a message was posted on a forum. Different advertisements are displayed because the applications randomly choose advertisements when they generate dynamic pages. Timestamps are different because the clocks on the two systems are not strictly synchronized (although it would not help if they did). These differences are normal variations across different application instances. They do not indicate any problems of the web applications. This confirms that Session Manager correctly handled all of the request instrumentations, because user requests in these experiments contain a mixture of GET and POST parameters, and HTTP cookies.

As discussed in Section 2.3, we use several heuristics to handle these normal variances so that they can be precluded from the list of potential problems that is presented to test engineers at the end of a testing period. The heuristics analyze differences in HTML page structures and the value distribution of the differing HTML nodes. For these applications, even though there are differences in response, HTML page structure stayed the same and the values of the differing node had a rather random distribution, and thus, Splitter had no problem suppressing these false positives.

Even though we suppress these differences as lower ranked, we nonetheless give an option for the test engineers to inspect them. In Table 1, we show the number of requests that were used to exercise each application, and out of them, the number of differences Splitter has found. Even though this is a relatively short test, we still found a large number of differences per application, which are too many to be useful to the test engineers if we present them as per URI level. We verify that Splitter is able to categorize these differences using Analysis Engine. In the Portal application, most of the differences are in the ads banner of the page *Default.jsp*. By looking at where these differences are within the pages, we can place them into the same category. Now, a test engineer only needs to check a few differences in each category to determine if the differences in the category are true errors or not. In the last column of Table 1, we show the number of problem categories Splitter has classified. This dramatically reduces the human efforts required to examine the differences. TPC-W has much more problem categories because jsessionid parameter strings appear in almost in every URI link and a page may have up to 58 such links in different places. However, by consolidating the cases to less than 300, it is fairly reasonable for test engineer to manually verify.

Application	Num. of Requests	Num. of Differences	Num. of Categories
TPC-W	118554	408320	277
Bookstore	5971	184	4
Portal	4527	272	2
Forum	3614	238	15
Yellow Page	2181	75	1

Table 1. Categorization of differences.

3.3 Working with Real Applications & Fault Injection

We have used Splitter in a few internal enterprise application migration projects. One of the migrated applications is IBM's Alphaworks which is a Web application for hosting trial projects and technical forums. We have encountered a few dozens of problems ranging from hardware problems to software errors. However, most of the problems are relatively easy to identify, e.g., hardware failure, because these problems either lead to the application not being able to launch or not performing at all. The more complicated problems are various configuration errors. These problems may only affect some of the requests or manifest themselves only under certain conditions. Splitter is particularly designed for such cases where the application is running but not producing all the correct contents. From working on several migration projects, we found a few categories of problems to be rather common during migration. One of them is related to files and access permissions, with problem descriptions such as "Missing jar files" and "Directories created with wrong permissions". Another type of common migration problem is related to authentication, with problem descriptions such as "errors with user ID management process" and "XYZ IDs are not functioning on server". Besides these configuration problems, we also noticed that database inconsistency and performance problems are also common causes for migration failure.

In order to illustrate how Splitter detects these common problems without releasing sensitive enterprise information, we manually inject different types of migration problems and evaluate Splitter's ability in detecting them. The injected faults affect different components in the system, including file system, application server, database server, and Web server.

For each application, we first deploy it on both platforms correctly. Then we manually inject faults to induce migration problems. After that, we replay the recorded user requests and use Splitter to detect the problems. Analysis Engine reveals a migration problem by showing the differences between the responses produced by the migrated platform and the original platform.

- **Type 1: Files and Access Permission**

One of the common mistakes during migration is not having all the needed files migrated and/or not correctly configured their access permission in the target system. In our experiment, we simply deleted some files or disabled read permissions of some files in the migrated system. As a result, when a client requests an affected file, the Web server will respond with an error code indicating the file is missing. This is easily identifiable by inspecting the HTTP headers. A sample Analysis Engine output of this type of problems is shown in Figure 11. The first part of the output shows the differences identified in HTTP headers (differences here are given a high rank), in particular, HTTP status code and content type fields.

```
Differences in HTTP Headers:
URI: /Portal/images/sign.gif
HTTP status: "200 OK" ==> "404 Not Found"
Content-Type: "image/jpeg" ==> "text/html"

Other Differences:
URI: /Portal/images/sign.gif
Different response bodies.
```

Figure 11. An example of missing files in *Portal*.

- **Type 2: User Credential and Authentication**

```
Differences in HTTP Headers:
URI: /bookstore/ShoppingCart.jsp
HTTP status: "200 OK" ==> "302 Moved Temporarily"
Location: " " ==> "http://NewPlatform/bookstore/Login.jsp"

Structural Differences in HTML Pages:
URI: /bookstore/ShoppingCart.jsp
-Node: html
```

Figure 12. An example of misconfigured client credentials in *Bookstore*.

Problems related to user credentials and authentication are also commonly found in Web application migration. They can be caused by a wide array of configuration mistakes, e.g., wrong SSL certificates, out-of-sync user credential information in databases, misconfigured trust policy between web server and application server. This type of problems usually cause the responses from the production and migrated applications to differ significantly in their page structures and have very distinct characteristics in HTTP header fields.

To simulate this type of problems on the migrated environment, we disabled HTTP cookies in Tomcat (by modifying context.xml). Splitter successfully detects the problem for all applications, except for *Forum*. Splitter does not detect any problem for *Forum* because it does not use cookies and thus it runs correctly on the migrated system with cookies disabled. For other applications, Splitter observes abnormal responses for the requests asking for user credential, such as shopping carts, user accounts, and administrative pages. For *Bookstore*, a client has to log into her account to check the items in her shopping cart. On the production system, the application can correctly identify the client with the cookie in her requests, and retrieve the 'shopping cart' belonging to this person. However, on the migrated system, the application cannot identify the client due to the missing cookie. Instead of returning a correct shopping cart page, the application sends the client a response to redirect the client to the login page (Login.jsp). This is realized by setting the HTTP status code and *Location* field in the HTTP header. The HTTP status code is set to "302 Moved Temporarily", and the *Location* field is set to the URI of the login page. Figure 12 shows the related console output of Splitter for this problem. Besides the differences in response headers, Splitter also shows structural differences in response bodies.

- **Type 3: Data Inconsistency**

```
Structural Differences in HTML Pages:
URI: /bookstore/ShoppingCart.jsp
*Node: html/body/table/
    Different Number of Children
...
```

Figure 13. An example of data inconsistency problem.

Migrating databases usually requires a team of specialized DBAs, and even so, this is an error-prone process especially when migrating a complex database setup. An error would cause persistent data to become different from that on the production environment, which in turn, causes the migrated application to behave differently. Such problems usually do not manifest themselves in response headers (as in case of the two above mentioned problem types), but rather in the response bodies. This problem is injected by populating two database instances differently. We show a part of Analysis Engine's console output for *Bookstore* in Figure 13. The difference is caused by the shopping cart pages (ShoppingCart.jsp) displaying different number of products due to the servlet populating the page with contents obtained from the database.

- **Type 4: System Resource Limitations**

Early in the migration process, the target system is sized and provisioned. However, if insufficient amount of resource is provisioned or configured, e.g., amount of physical memory, open file descriptor quota, disk quota, etc., the migrated application will run into both functional and performance problems, especially under heavy load. To simulate such problems, we reduce the size of the buffer cache used by PostgreSQL in TPC-W, and we reduce the maximum network connections (using ulimit command) for the other applications. For TPC-W, without sufficient system resources, database server becomes a bottleneck. Splitter will eventually detect this problem by monitoring the queue length of the replicated requests. Due to the performance difference between the production and migrated versions of the application, this queue will kept increasing under heavy load until it hits a maximum allowable queue length, at which point, we terminate testing as it is apparent that the migrated application cannot keep up with the pace of the production application. As for the other applications, Splitter will identify these resources problems when it intermittently receives "Internal Server Errors" when the application server is running out of resources.

Splitter also supports rudimentary performance monitoring as it can easily collect response times of all the traffic. Due to space limitation, we will not discuss this capability in details. However, combined with the functional testing capabilities of Splitter, it can be quite powerful in detecting some hard-to-reproduce performance problems.

3.4 Performance and Storage Overheads

In this section, we evaluate Splitter's performance and storage overheads. In Splitter, costly operations are to find detailed differences in responses. These operations are carried out off-line. Thus, sources of performance impact only include extra network hop between the production system and clients and computation used to replicate and instrument requests. For HTTPS traffic, it also incurs extra overheads on encryption and decryption to see all traffic in clear text. Storage requirement is because Splitter stores some request and response data for off-line comparison.

We evaluate Splitter performance impact using the TPC-W workload for both HTTP and HTTPS traffic. Among the three types of workloads defined in the *TPC-W* specification, we select its browsing workload to stress Splitter as it incurs the highest number of transactions.

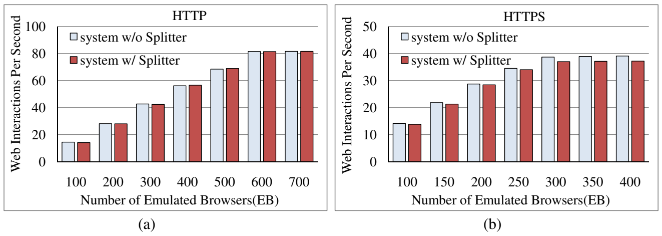


Figure 14. Throughputs of *TPC-W* observed by clients with and without Splitter. The number of emulated browsers (EBs) is varied from 100 to 700 for HTTP traffic (sub-figure a) and from 100 to 400 for HTTPS traffic (sub-figure b).

The throughput results are shown in Figure 14. Results are shown for clients retrieving web contents with and without Splitter. In both scenarios, we collect throughput data for both HTTP and HTTPS traffic as we increase the workload by increasing the number of emulated browsers (EBs) till the production system is saturated.

As shown in Figure 14, Splitter incurs only a small performance overhead, even for HTTPS traffic. The throughput reduction is almost negligible (less than 2%) for HTTP traffic and is less than 5% for HTTPS traffic. The reason is that the delay and extra processing time that Splitter introduces is much smaller than the time it would take for the application server to service a request. Specifically, it takes the application and Web server over 20ms on average to service a TPC-W request even when the system is under a light load (100 EBs), which is much longer than one network hop latency (less than 100μs in our local network) or the time used to replicate a request. Splitter incurs more performance overhead for HTTPS traffic due to the extra work needed to encrypt and decrypt HTTPS traffic.

To evaluate the storage overhead incurred by Splitter, we collect the size of the data files used by Splitter to store request and response data for off-line comparison. On average, the storage overhead for *TPC-W* is about 14.3KB per web interaction without compression and about 1.6KB with compression. One million web interactions, which are more than enough to test an application, require less than 2GB space. Considering the TB level of storage volume in modern hard disks, the storage requirement of Splitter is quite reasonable.

4. Discussion

In this section, we discuss some of our deployment experience and a few limitations of our current prototype and propose possible solutions. Work is currently underway to incorporate these solutions in our implementation.

4.1 Deployment Considerations

4.1.1 Deployment Requirements

In our experience helping migration teams performing post-migration testing, we learned that besides being useful, Splitter also needs to be easily deployed and have minimal impact to the existing environment. Specifically, Splitter needs to meet the following three requirements.

- Minimum intrusiveness to existing infrastructure: Splitter taps into the critical path between end users and the production application, and therefore, would require some changes to the existing infrastructure and even some service down times, which should be minimized.
- Minimum performance impact: Splitter should not introduce significant performance degradation to the application perceived by end users, though some overhead is unavoidable since Splitter will be positioned within the critical path.
- User transparency: End users should not be aware of the existence of Splitter. This means no changes to the end user settings and no service disruptions.

User transparency and minimal performance degradation can be easily achieved. Since Splitter is deployed as a reverse Web proxy that we embed in front of the application server, it simply adds one extra hop in addition to many other hops (network proxies, load balancer, switches, etc.) along the path from end users to the application. However, having a minimum impact on the existing infrastructure when deploying Splitter is more challenging. We discuss the different deployment options and their pros and cons below.

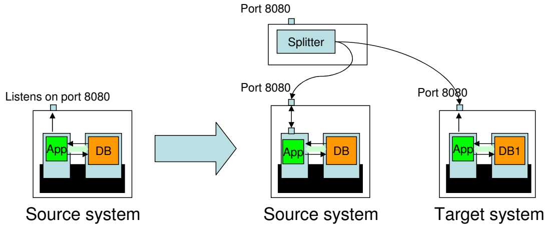


Figure 15. Deployment option 1.

4.1.2 Deployment Options

Option 1: The first deployment option is illustrated in Figure 15. Here we deploy Splitter onto its own system. The greatest benefit of this approach is everything is self-contained, and can be delivered as a (virtual) appliance. However, there are several drawbacks with this approach. First, the new system Splitter is on, whether it is physical or virtual, will need its own IP address. And to insert it into the path between the user and the application, system administrators would need to modify the network forwarding path, firewall rules, SSL certificates, and sometimes even DNS entries or IP addresses. That would take a non-significant amount of time as these tasks often involve multiple system admin groups, especially in a large enterprise. To reduce the time taken for the deployment, the databases on the two systems can be synchronized online with tools such as Data Mirror [Ghatare] and Splitter can be pre-configured before dropped into the environment. Leveraging virtualization technology makes this type of deployment even easier as virtual machines can be provisioned and taken down quickly and with low cost.

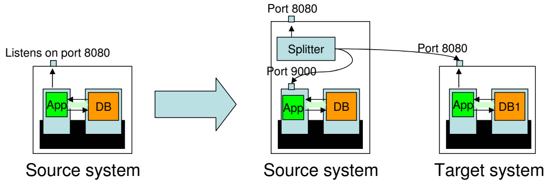


Figure 16. Deployment option 2.

Option 2: The second deployment option is shown in Figure 16. In this alternative approach, Splitter is co-located with the production application on the same machine. There are obvious cost and management benefits in not instantiating a separate machine to host Splitter. Additionally, having Splitter listens to the same IP address and port of the production application and reconfiguring the application to listen on another port, no network reconfiguration is needed as in the case of Option 1. Moreover, the performance overhead of the "extra network hop" that Splitter introduces when co-located with the production application is also lower.

From our perspective, it seems Option 2 is the better deployment architecture. However, in practice, the first Option is more preferred because the changes in the production environment would only involve system administrators, whereas Option 2 would not only involve system administrators but also application owners as Splitter is co-deployed on the same machine as their application.

In enterprise environments, it is common to use load balancers to distribute network traffic going to the destination web servers. Usually the existence of load balancer implies that the server may consist of multiple nodes and may support multiple applications. In such an environment, the deployment of Splitter is a little more complicated. As one choice, we can deploy Splitter in front of the load balancer or integrate Splitter (only the Proxy and Session Manager) within the load balancer. In this case, we only need one instance of Splitter. But Splitter needs to detect and filter traffic going to other web applications instead of the application under test. If we deploy Splitter behind the load balancer, we need one instance for each of the server node. This deployment may require more physical resource but traffic handling is simpler.

4.2 Out-of-Order Requests

A web application is usually designed to support multiple concurrent users. In our testing infrastructure, there are 3 separate streams of request-response traffic—(A) user to Splitter, (B) Splitter to production application, and (C) Splitter to migrated application. Ideally, we would like to preserve the order of requests in stream A to those of B and C. However, this is sometimes not possible. For example, if user U1 sends a request R1 and user U2 sends a request R2, respectively, to Splitter, and then Splitter forwards them to the production and migrated applications, R1 and R2 might reach the two applications in different orders.

For some types of applications, servicing out-of-order requests does not matter, e.g., web forums, social networking sites, etc., however, for other applications such as banking and online shopping, the order of requests is crucial to preserve. Since we have done all our experiments in a LAN environment, we have not seen any problems due to out-of-order requests. One solution to this problem is to enumerate the requests as they arrive at the Proxy and have a small process that marshals the requests to the production and migrated applications according to the order they arrived.

4.3 Visibility of Client IP Addresses

In some cases, client IP addresses are needed for purposes such as logging and authentication. However, due to the intervention of Splitter, client IP addresses are not available directly to Web applications. There are two methods to solve the problem, each of which has its advantages and disadvantages.

One method is that Splitter passes client IP addresses to Web application with the HTTP X_FORWARDED_FOR header field, which is widely used by Web proxies and load balancers to pass client IP addresses to the Web servers behind them. However, this method requires an application to handle the X_FORWARDED_FOR header field to get client IP addresses. While most applications behind proxies or load balancers meet the requirement, other applications may need some minor modifications to handle the X_FORWARDED_FOR field in order to apply this method.

The other method is to carry out logging and authentication on Splitter as client IP addresses are available to it. Splitter is based on a fully-functional Web proxy. It can be configured to log client IP addresses and other request information. When testing finishes, client IP addresses logged by Splitter can then be used to replace those on the production and migrated systems after appropriate format conversion. As for authentication, the access control list (ACL) of Splitter can be configured to specify which IP addresses are allowed by the application and which are not, so that only requests from the allowed IP addresses are forwarded and replicated by Splitter. This method does not require the application to handle the X_FORWARDED_FOR header field. However, the lists of IP addresses that are allowed or denied should not be too complex to be described with the ACL rules of Splitter. At the same time, because client IP addresses are still unavailable to Web applications with this method, it can hardly be applied in scenarios other than logging or authentication.

4.4 Non-HTTP Channels

Other than HTTP traffic, some Web applications also generate and make use of other types of traffic, e.g., an online store application generates a confirmation email after a client has placed an order. Not being able to monitor SMTP or SMS traffic prevents us from fully testing all functional areas of an application. However, these types of traffic can be interpositioned similarly by using the proxy approach we have shown here for HTTP traffic. Our implementation described in this paper is not meant to be an all-encompassing solution, but rather one particular implementation (specifically for Web application) of a testing framework. This implementation can be expanded to handle more complex application types and protocols.

5. Related Work

5.1 Sandbox Validation

Maintenance and configuration tasks carried out by system administrators may greatly impact system availability and performance. It is important to validate these tasks to guarantee that the works performed have achieved the desired results (e.g. better functionalities and/or performance) and not resulted in unexpected downtimes or performance degradation. For this purpose, several previous works [Nagaraja 2004, Oliveira 2006, Zheng 2009] have been conducted to create a sandbox to run a copy of the system to be changed. Instead of working on the original system, administrators work on its replica inside of a sandbox. To drive the replica, inputs are replicated and directed to the sandbox. To validate the operations carried out by the administrators, outputs of the replica are compared against those produced by the original component.

Splitter also replicates inputs and compares outputs. In this sense, testing with Splitter is similar to sandbox validation. However, a sandbox is usually tightly coupled with other components in the system. To apply sandbox validation requires much modification or reconfiguration on the current system. This makes it difficult to be applied to the post-migration scenarios. Our Splitter infrastructure is designed to minimize the intrusiveness to the current system infrastructure.

5.2 Web Application Testing

Since Web applications have been widely used to support a range of important activities such as online banking, online store, etc., Web application testing has attracted much attention. Basically, there are two approaches to test and validate a web application. The first approach focuses on analyzing and modeling the structure of a web application and helps tester with coverage testing [Lucca 2002, Ricca 2001]. However, this approach needs testers to generate test cases, especially to prepare inputs for test cases. Moreover, human inspection is usually needed to check the outputs of the tests. The other approach records user requests and user sessions, and transforms user sessions into test cases [Elbaum 2003, Sampath 2004]. This approach also needs a lot of manual intervention to check the outputs of test cases. The research presented in paper [Sampath 2004] has some similarity to our work, but test cases used in that work are still synthetic and a simple "diff" is used for comparison.

The above techniques designed for web application testing are mainly for checking the programming errors or bugs before the applications are deployed. However, Splitter is designed to test applications after they have been deployed, targeting provisioning errors and configuration errors.

5.3 Change Detection

There have been research works done designing efficient algorithms to detect changes in text files [Chawathe 1996] and structured XML and HTML documents [Cobena 2002, Khoury 2007, Leonardi 2005, Ohst 2003, Wang 2003]. One of the challenges in our testing module is to detect response changes in the migrated application. Thus the algorithms (e.g. SES algorithm) developed in the area of change detection can be leverage by Splitter. But we still need to fine-tune them to filter false alarms.

5.4 Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) [Castro 1999] has been extensively studied to prevent malicious network attacks and software errors. BFT requires all replicas to execute an identical sequence of client requests. In this sense, Splitter is similar to BFT. However, BFT requires more than three replicas to make consensus. Moreover, it usually requires non-faulty replicas to produce same responses for same request. In contrast, Splitter deals with only two replicas. For the same request, they may produce different responses because web applications usually exhibit sophisticated non-determinism.

6. Summary and Future Work

In this paper we have described a new approach for post-migration testing of Web-based applications. Our approach uses the actual user workload in real-time to compare responses from the migrated application against those from the original production application before cut-over to the new platform. Our approach offers an appealing alternative to the traditional FVT-based testing method which requires considerable time and effort to develop and execute test cases.

We have implemented our approach in Splitter, a software module which consists of a modified reverse *Proxy*, a *Session Manager*, and an *Analysis Engine*. User requests are duplicated and forwarded to both of the production application and the migrated application. Besides being forwarded back to the user, responses from the production application are compared automatically with those from the migrated application.

We have evaluated our prototype of Splitter using several Web applications as migration candidates. While Splitter does detect errors that are in fact normal, Analysis Engine is able to group mismatches that are highly likely to have the same root cause, which reduces the number of cases that a tester must inspect. We also created a number of fault injection cases that represent some of the errors observed in actual migrations – Splitter is able to effectively detect the errors we considered.

The complexity and overhead of Splitter is similar to those of a Web proxy or a load balancer. Our evaluation with *TPC-W* shows that Splitter incurs only small performance and storage overhead. In Splitter, costly operations are to find detailed differences between HTTP responses. These operations are carried out offline. Operations that are carried out online are similar to those in a Web proxy or a load balancer, e.g., HTTP header parsing, URL rewriting, and cookie handling. Splitter can be integrated with a Web proxy or a load balancer that has already been deployed in a production system to further reduce system complexity and overhead. This can be our future work.

As a part of our ongoing work, we are continuing to improve Analysis Engine. We want to test and improve its detection algorithms by injecting additional classes of migration faults. We also plan to improve Analysis Engine to make it accept plugins that handle additional response content types. With new plugins, we want it to be able to dissect and to compare binary objects dynamically generated by Web applications. Future Web applications may generate responses with some new types of contents. To analyze and to compare them also require Analysis Engine to be easily extended to cover new content types. We also plan to apply Splitter to test migrations of more complex enterprise applications which will potentially uncover new classes of errors. Finally, we are implementing the solution outlined in Section 4 to handle request ordering issue during the testing phase.

7. Acknowledgements

We are grateful to Dr. Emmanuel Cecchet for his detailed comments and suggestions on the final version of the paper. We thank the anonymous reviewers for their constructive comments. We would like to acknowledgement helpful discussions with Dr. Kamal Bhattacharya and Dr. Sambit Sahu. This work is partially supported by National Science Foundation grants CCF-072380. CNS-0834393, and CCF-0913050.

References

[AlertSite] AlertSite. Dejaclick by AlertSite. `http://www.dejaclick.com`.

[Benedikt 2002] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic Web sites. In *WWW '02*, 2002.

[BMC] BMC Software Inc. BMC discovery. `http://www.bmc.com/products/offering/bmc-discovery.html`.

[Castro 1999] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99*, pages 173–186, 1999.

[Chawathe 1996] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[Cobena 2002] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *ICDE '02*, 2002.

[Elbaum 2003] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *ICSE '03*, pages 49–59, 2003.

[Ghatare] Indrani Ghatare. Using WebSphere DataStage with ibm DataMirror change data capture. `http://www.ibm.com/developerworks/edu/dm-dw-dm-0805ghatare-i.html`.

[Hayes 2008] Brian Hayes. Cloud computing. *Commun. ACM*, 51 (7):9–11, 2008.

[IBM a] IBM. Tivoli application dependency discovery manager. `http://www-01.ibm.com/software/tivoli/products/taddm/`.

[IBM b] IBM. Tivoli provisioning manager. `http://www.ibm.com/software/tivoli/products/prov-mgr/`.

[Khoury 2007] Imad Khoury, Rami M. El-Mawas, Oussama El-Rawas, Elias F. Mounayar, and Hassan Artail. An efficient Web page change detection system based on an optimized hungarian algorithm. *IEEE TKDE*, 19(5):599–613, 2007.

[Leonardi 2005] Erwin Leonardi and Sourav S. Bhowmick. Detecting changes on unordered XML documents using relational databases: a schema-conscious approach. In *CIKM '05*, 2005.

[Lucca 2002] Giuseppe A. Di Lucca, Anna Rita Fasolino, Francesco Faralli, and Ugo de Carlini. Testing Web applications. In *ICSM '02*, pages 310–319, 2002.

[Marden 2001] Morris Marden. An architectural evaluation of java TPC-W. In *HPCA '01*, page 229, 2001.

[Nagaraja 2004] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04*, 2004.

[Novell] Novell Inc. PlateSpin migrate. `http://www.novell.com/products/migrate/`.

[Ohst 2003] D. Ohst, M. Welle, and U. Kelter. Difference tools for analysis and design documents. In *ICSM '03*, 2003.

[Oliveira 2006] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and validating database system administration. In *USENIX'06*, 2006.

[Ricca 2001] Filippo Ricca and Paolo Tonella. Analysis and testing of Web applications. In *ICSE '01*, pages 25–34, 2001.

[Sampath 2004] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. Composing a framework to automate testing of operational Web-based software. In *ICSM '04*, Washington, DC, USA, 2004.

[Squid] Squid Developers. Squid proxy cache. `http://www.squid-cache.org/`.

[Wang 2003] Y. Wang, D.J. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In *ICDE '03*, pages 519–530, 2003.

[Williams 2005] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In *Web Content Delivery*, pages 3–21. Springer, 2005.

[Wood 2000] Lauren Wood, Vidur Apparao, Steve Byrne, Mike Champion, and Scott Isaacs. Document object model (DOM) level 1 specification (second edition), 2000. `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[Young 1977] Ian T. Young. Proof without prejudice: use of the kolmogorov-smirnov test for the analysis of histograms from flow systems and other sources. *The Journal of Histochemistry and Cytochemistry*, 25(7):935–941, 1977.

[Zheng 2009] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, J. Renato Santos, and Yoshio Turner. JustRunIt: Experiment-based management of virtualized data centers. In *USENIX'09*, June 2009.