A Design Pattern to Decouple Data from Markup

Balwinder Sodhi and T.V. Prabhakar

Department of Computer Science and Engineering
IIT Kanpur, UP India 208016
{sodhi,tvp}@cse.iitk.ac.in

Abstract. Modern web applications have steadily increased in richness and complexity, and they put significant demands on system resources such as server CPU, memory and most importantly the network bandwidth. When seen at Internet scale, a tiny wastage in a resource can translate into a huge loss. For instance, we will show that youtube.com homepage can potentially save up to 4500 GB worth of bandwidth every day! It is, therefore, important for the application designers to: 1) identify what opportunities exist for improvement and 2) ensure that computing resources are efficiently utilized.

We present the results of an extensive investigation of how the *useful information* is distributed across various HTML tags and their attributes inside the served HTML pages taken from a large number of dynamic public websites. Our findings show that the *useful information* is often restricted to only a handful of the tags and attributes. We systematically explore the efficiency differences between various classes of frameworks that are used for developing modern web applications. Leveraging our findings, we propose a technique which decouples the view's markup and data thus allowing them to travel separately and only *on demand*. This improves the web application efficiency; for instance our experiments show that this approach increased the throughput by a factor of about 7.

1 Introduction

With the web browser increasingly becoming the powerful and popular platform for delivering business applications and services, the need to identify the opportunities that may exist for improving various aspects of the web applications design, therefore, becomes important. Typically, web applications are accessed via a browser that interacts with the application server via HTTP to fetch HTML content to be displayed in the browser. On the application server side, the web application may be serving just the static HTML pages; or, as in majority of the business web sites, it could be serving the dynamic content. Dynamic content is often created by combining *data* with the *templates* on the server. The *data* part is what we'll call the *useful information*. Often the *useful information* is the directly visible content on a web page in response to a specific query submitted by the user. For instance, on a shopping website's product catalogue page the

C. Huemer and T. Setzer (Eds.): EC-Web 2011, LNBIP 85, pp. 25–38, 2011.

information about each displayed product item is the useful information. The *template* is serving mainly as the presentation vehicle.

The problem with most dynamic web applications is that they often serve the final render-ready HTML to the browser. This is wasteful in terms of network bandwidth consumed, server side CPU and memory consumption etc.

Processing flow in most dynamic web application development frameworks follows these high level steps:

- Query the data from some data store like a relational database engine.
- Load and parse the template for the response page to be served. Often this results in some sort of object graph to be created to represent the page in memory.
- Go over the parsed page template and fill into it the data that was fetched from data store.
- Marshall the data-hyderated template as HTML and send it out as the response.

Clearly, all of this needs to happen on the application server and consumes server CPU and memory. We are interested in improving the situation. We examine how *useful information* is distributed inside HTML page elements in order to get insight into what changes in the content from one request to another for a given page. We make use of the empirical data that we derive from our experiments on large number of web applications to propose mechanisms which could result in better utilization of the computing resources.

Rest of this paper is organized as: section 2 describes our methodology that we followed to investigate the HTML pages and discuss the findings. In section 3 we provide relevant background on the working of popular web application frameworks. Section 4 we describe the proposed approach and discuss the experimentation results.

2 Investigation Methodology and Results Analysis

2.1 Selection of Web Applications

We focussed our study on the data driven dynamic websites. Our selection of the web sites was based on the Alexa's top websites list in the shopping category [1].

We selected this list because, first, it represents a fairly big sample population and thus provides an accurate and representative sample of current Internet sites in this category. Secondly, this list represents the most popular Web sites, based upon traffic information [2]. This *popularity* is derived from the page views, number of pages viewed on a host, and the reach (number of different users who access a host). The Alexa ranking is a based on the geometric mean of the reach and views quantities.

2.2 Data Collection

To automate the process of data collection we developed a tool to capture the statistics about the distribution of useful information in an HTML page as it

Table 1. Meaning of various HTML statistics

Statistic	Remarks
Tag	Name of the HTML tag
Attribute	Name of attribute of the tag
Attribute size	Size of the attribute value in chars
Attribute size ratio	Ratio of attribute size (in chars) to the total HTML file size (in chars)
Attribute size compression ratio	Used as an indicator of content repetitiveness in the attribute value
Tag count	How many times the tag appeared in the HTML
Text size	Size of the text content of the tag in chars.
Text size ratio	Ratio of tag's text size (in chars) to the total HTML file size (in chars)
Text size compression ratio	Used as an indicator of content repetitiveness in the tag's text value

is served on the browser. From the completeness of the analysis purposes we consider the entire set of HTML elements and their attributes when looking for what data they contain. In practice, however, only the data present in the following places on the HTML is found to be useful:

- Text inside the body of an HTML tag. For instance, text inside the body of `DIV, P, SPAN, A` etc. tags is often found useful.
- Value of the HTML tag attributes (e.g. `src, id, value` and `href` etc.).

We execute the data collection tool on the identified set of web applications and capture the statistics about HTML elements. Table-1 shows what each of the captured statistics mean.

2.3 Results Analysis

We analyse here the results from our study of content distribution inside HTML pages. Based on the insights gained from this analysis, we will look into the opportunities for improving the efficiency of certain web applications in the subsequent sections.

Table-2 shows[1] the overall distribution of the information between the HTML tag's text and attribute values for some of the top shopping web sites. Amongst the entire set of web sites that we studied, the maximum %age of data contained in the tags text was observed to be about 60%, minimum was at about 10% and average was about 24%. Similar numbers for the data contained in tag attribute values was: maximum at about 60%, minimum at about 15% and average was about 36%. All these %ages are w.r.t the total size of the served HTML content.

Another important set of statistics is the attribute-wise distribution of information as shown in the Table-3. We notice that the largest contributor here is the `href` attribute of the `A` tag for majority of the websites. Second largest

[1] For want of space we are showing, in tables 2, 3 and 4, data for a small subset of web applications that we studied.

Table 2. Overall information distribution

Tag Text (chars)	**Attr. Text** (chars)	**% Data in Tags**	**% Data in Attr.**	**% Data Total**	**Host URL**
23084	67402	11.48	33.51	44.99	www1.macys.com
41171	85900	16.4	34.22	50.62	www.overstock.com
8109	33171	9.97	40.8	50.77	www.target.com
33603	31935	26.88	25.55	52.43	electronics.shop.ebay.in
11236	25066	17.14	38.24	55.38	autos.yahoo.com
16552	44228	15.19	40.59	55.78	www.tigerdirect.com
11658	43403	12.75	47.48	60.23	www.staples.com
53107	150646	16.03	45.47	61.5	www.alibaba.com
17707	66818	13.7	51.69	65.39	www.barnesandnoble.com
45130	70180	25.91	40.29	66.2	www.buy.com

contributor is the `src` attribute of the `IMG` tag, even though its relative contribution as compared to the top contributor is small (about one third on average). Contribution of all others is less than 5% individually. We also observe that the compression ratio for these top contributors is fairly low (about 0.2) and is an indicator of a fair amount of repeated text there. We investigated these low compression ratio cases further by looking at the actual HTML content of those cases. We observed that the most `href` values had a full URL of which a large part (often upto the path component) was common and only the small query component was changing. Situation with the `src` attribute of the `IMG` tag was very similar. One such example is shown in Listing-1.1 where the repeated URLs are shown in bold underlined font. Further, we analysed the distribution of information amongst the text content appearing directly inside the body of various tags in the served HTML content. These statistics are shown in the Table-4. The largest contributor here is the `SCRIPT` tag whose max contribution in one case is about 55% towards the total size of HTML served, the minimum contribution is about 3% and on average it contributed about 15%. Other top contributors are the `P` (average contribution about 1%) and `A` (average contribution about 3%) tags. Average compression ratios in these cases is relatively high: 0.3 for `SCRIPT`, 0.52 for `P` and 0.4 for `A` tag. This indicates that there's not much repetitiveness in the information contained in these tags. In summary, all these findings indicate the following about the distribution of information in a served HTML page:

- Major chunk of the *useful information* is held in small number of tags and/or their attributes, which often are: `A/href, IMG/src, SCRIPT, P`.
- There is often a fair amount of repetitiveness in the information held by those tags/attributes that are largest contributor towards the HTML page's size.

We'll leverage the findings of our investigation as discussed in this section to propose a technique to improve the web application efficiency.

Table 3. Attribute-wise information distribution

Tag	Attribute	Attr. Size (chars)	% Data in Attr.	Compression Ratio	Host URL
A	href	70088	41.389	0.212	finance.yahoo.com
A	href	31567	34.532	0.215	www.staples.com
A	href	30918	23.916	0.154	www.barnesandnoble.com
A	href	24585	22.561	0.156	www.tigerdirect.com
A	href	32955	18.921	0.195	www.buy.com
A	href	11970	18.262	0.145	autos.yahoo.com
A	href	40645	16.19	0.148	www.overstock.com
A	href	11049	13.591	0.222	www.target.com
A	href	41571	12.547	0.149	www.alibaba.com
IMG	src	15418	9.105	0.135	finance.yahoo.com
A	href	17387	8.645	0.103	www1.macys.com

Table 4. Tag-wise information distribution

Tag	Occurrence	Text Size (chars)	% Data in text	Compression Ratio	Host URL
SCRIPT	7	32474	55.312	0.196	www.istockphoto.com
SCRIPT	43	35008	37.377	0.184	www.walmart.com
LI	33	1508	22.086	0.253	www.netflix.com
SCRIPT	43	36636	21.035	0.218	www.buy.com
STYLE	2	928	13.591	0.48	www.netflix.com
SCRIPT	21	16659	13.328	0.307	electronics.shop.ebay.in
A	611	9381	10.262	0.354	www.staples.com
SCRIPT	41	33529	10.12	0.274	www.alibaba.com
OPTION	1291	11178	8.943	0.375	electronics.shop.ebay.in
SCRIPT	25	5827	8.89	0.334	autos.yahoo.com
SCRIPT	46	13054	7.709	0.266	finance.yahoo.com

Listing 1.1. HTML fragment showing repetitive content

```
<map name='1307012650' id='1307012650'>
  ...
 <area shape='rect' coords='2,89,177,153'
   href='http://www.target.com/gp/browse.html/ref=sc_iw_l_0_2/?node=10218751'
   alt='TVs.' />
 <area shape='rect' coords='2,155,186,209'
   href='http://www.target.com/gp/browse.html/ref=sc_iw_l_0_3/?node=3666481'
   alt='Baby Furniture.' />
  ...
</map>
```

3 Web Application Frameworks - A Literature Review

Following are the major frameworks which are used to build the *core* part of web applications that are developed for various platforms. Recent releases of these frameworks provide support of modern Web 2.0 technologies such as AJAX [3,4] and JSON [5] etc. which can be leveraged to compose efficient solutions. In this paper, these frameworks are grouped into three categories based on how they support building the web UI. Items in each category are not exhaustive; since all frameworks in a given category are conceptually similar therefore only few popular frameworks are listed under each category.

3.1 Frameworks Requiring a Plug-in and/or Scripting Engine in Browser

Frameworks in this category allow creating Rich Internet Applications (RIA) which are aimed at providing the desktop application like user experience. The application is often an executable which is downloaded into the browser and executes via a special scripting engine/plug-in inside the browser. Interactions with the application server are often via RPC style AJAX calls. Popular frameworks in this category are: Google Web Toolkit [6], Microsoft Silverlight [7]

Issues with this Category. Following are the major issues with GWT/Silverlight kind of frameworks,

1. Major part of the UI has to be delivered as a single big script (e.g. .xap for Silverlight and .js files for GWT) which executes within the browser. Though there are different tricks and techniques which can allow a developer to reduce the initial startup time for the application, but the fact remains that for any serious business application the initial download and startup time can be unacceptable, especially over a limited bandwidth connection.
2. Restriction on what can be achieved with a browser-side scripting engine can sometime be a limiting factor for some usecases. For a public facing web application, the assumption about a scripting plugin being available and enabled in the browser may not always be realistic. Heavy reliance on JavaScript (e.g. in GWT like framework) can lead to cross-browser compatibility issues. For instance, in GWT although the code development in Java eases the programmer's effort, however, it is relatively difficult to troubleshoot the generated JavaScript code that finally executes in the browser.
3. Silverlight needs a special browser plug-in in order to execute the application. While this may not be a major issue in newer client machines,for older machines it may be a limitation.
4. Such frameworks use very specialized programming models, hence there is often a significant learning curve and maintenance costs associated.
5. It is difficult to prevent divulging the application code and logic. In case of scripting code based applications the code gets downloaded to the client.

3.2 Frameworks Based on Server-side Tag Libraries and Scripting

Typically, the web application here employs a template engine (e.g. Python Django, Apache Velocity) to combine the data with the presentation markup to create the HTML pages to be served to a browser. In many cases the markup may be mixed with the scriptlets as in Java Server Pages, PHP pages etc. to produce the dynamic HTML pages to be served. Popular frameworks in this category are: Apache Struts [8], Ruby on Rails [9], PHP Zend [10]

Issues with this Category. Most issues in this approach of web application development are related to both server side memory and CPU consumption and more importantly the content sent across the network. Following are the main steps (also shown in Figure-1) that happen on the server for producing a response to a requested dynamic page:

1. Every time a page is requested by the client, the server has to first parse the page (e.g. .jsp or .aspx page etc.) markup to determine what dynamic scripts/tags needs to be instantiated.
2. Instantiate the objects for and execute the scriptlets and tags found on the requested page. This step also performs the data binding, if any, onto the resulting markup.
3. Generated markup which will also contain the data for various UI elements is then sent over the network to the client. Often in the flow of an application, only the data changes between requests to the same page.

Thus the above steps put unnecessary load on the server as well as the network. All that the client needs from server is new data which the client can refresh its display state with.

3.3 Framework Using Component Based User Interface

Here the UI on the server-side is object oriented and created by composing an object graph similar to how it is done for the desktop applications. For instance, a view has a page object and page may have many panel objects and each panel may have various widget objects like text box, label and buttons etc. This component tree is usually rendered as HTML response. Popular frameworks in this category are: Java Server Faces [11], ASP.NET Web Forms [12]

Issues with this Category. In this approach as well the issues are exactly same as in the case of server-side scripting/tags.

4 Exploiting Web 2.0 Technologies — A Hybrid Approach

In each of the frameworks mentioned in previous section it is possible to make use of Web 2.0 technologies such as AJAX, JSON and client-side scripting tools etc.

We exploit the findings of our investigation discussed in section 2.3 to address the issues that have been mentioned for the popular web application frameworks and improve the efficiency. One such technique is discussed below. The key ideas underlying this approach are:

- Separate the data and markup (representing the UI) at the *view* level. Important thing here is the granularity of the *view* (i.e. how many standard UI elements/widgets are packed in the view's markup). Often it is application dependent and can be easily controlled and tuned. The data and the markup of the UI view can travel separately and only *on demand*. Application efficiency increases.
- Use static or one-time dynamically generated markup to create the views in UI. It requires much less server memory and CPU cycles as compared to traditional component/tag-library based UI because it won't be creating the document tree or the object graph on each request for representing the view. Also, manipulation of plain HTML using JavaScript on client side requires cheaper programming skills.
- Use lightweight data interchange mechanism such as JSON to exchange data between client and server.
- The content – both markup and data – can be served by any of the existing server-side framework.

This approach works with plain HTML and JavaScript to create *templatized views* for web UI. A *view* here can be a complete screen (full HTML page) or a section (HTML fragment) of it. These *view templates*, which are plain HTML markup without any data in the start, are incrementally brought to the client as and when they are needed by the user. Any subsequent activation (in a single session) of a view doesn't result in bringing the markup again to the client – only data is fetched from the server as needed. Data binding to UI elements is done on the client-side via DOM manipulation using JavaScript. Following sections provide further details. Request-response interactions occurring in the proposed solution are depicted in Figure-2. For comparison purposes, similar interactions, as they happen in tag-library based approach (which is similar to UI component based approach), are shown in Figure-1. We'd like to mention that this technique does not try to replace the existing frameworks such as those listed in section-3; instead it is aimed at complementing their capabilities to satisfy special efficiency requirements and constraints.

4.1 Use a Compact Data Interchange Format

The data interchange format selection can be based on the following broad criteria:

- For a given amount of information to be exchanged, it should produces a very compact payload.
- Should be easy to deserialize/unmarshal via client scripting tools.
- Availability of very good quality libraries for both client and server side use.

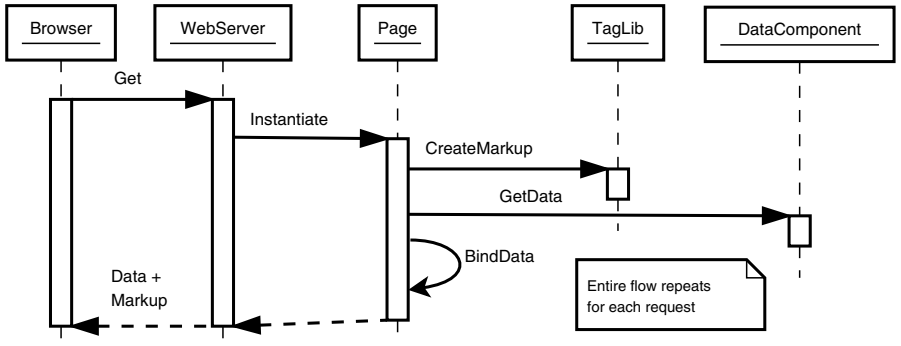

Fig. 1. Request response interactions in tag-lib based approach

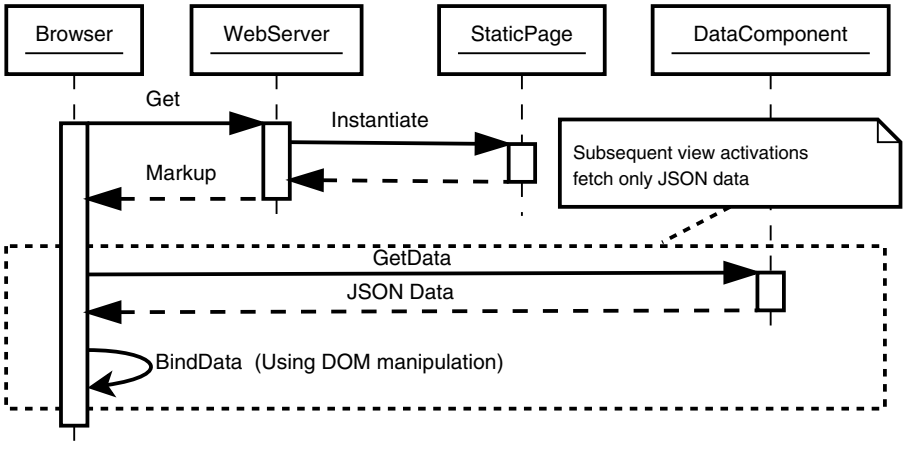

Fig. 2. Request response interactions in proposed solution

One reasonable candidate for web applications is JSON. Use of JSON along with templatized views (as explained in next sub section) helps in reducing the data flowing between client and server. In any session, application views are fetched only once from the server as and when needed. Views are added/updated on the page using DOM manipulation. Data for the view is fetched separately in JSON format. Any subsequent activation of a view requires just the data part to be fetched from server. This results in reducing unnecessary ferrying of "HTML tags" between client and server, thus improving the efficiency of the application.

4.2 Use Static Template Based Views to Construct UI

View template here refers to a piece of HTML markup which defines a screen or its section. Minimal markup is used just enough to indicate the arrangement of UI elements on the view. Assigning proper IDs to different elements is important for later data binding via DOM manipulation using JavaScript. For example, if a

view contains a text field and a table containing 4 columns, then the markup will contain just an INPUT tag and a TABLE tag of HTML. Since view templates are static, so no dynamic server-side objects are created when client requests the page/view. This static markup is also cacheable, and even content delivery networks could be leveraged to further improve the performance.

How data is bound to UI elements. Each data-bound HTML element in a view template is assigned a unique ID. The data-bound HTML element also has an attribute to specify which JSON object property holds the data for this element. JSON data fetched from server is bound to UI elements via DOM manipulation using JavaScript. The approach is similar to [13, Ch. 3], except that in our approach we are getting only the data as JSON. There are decent client-side scripting libraries (e.g. JQuery [14], DOJO [15], YUI [16] etc.) available which can ease this task.

Effective use of AJAX. It is common in many web applications to make use of AJAX calls from the client to server for fetching either just the data or the final render-ready HTML hydrated with necessary data. The AJAX interactions between client and server in such a case are at basic UI element level (e.g. a dropdown list of a check-box) and thus are too fine grained. For example, on checking a check-box the application may re-calculate (on server-side) the values shown on some textboxes. Perhaps in some use cases this kind of fine grained interactions is what the application needs. However, it can potentially result in a client being too chatty with the server. If the concurrent user base for the application is large then such a chatty client introduces additional connections load for the server and can be problematic. In the cases where render-ready data-hydrated HTML is fetched over AJAX from the server, the problem is essentially same as described for traditional tag-library based approach. This is because, though the data + markup are brought over AJAX, but still the data *and* markup travel together. In the AJAX technique discussed in this paper we decouple the data and markup at the *coarse grained view* level. Hence it avoids both the above issues as observed with plain use of AJAX.

Trade-offs involved

1. Input validation on server-side will become somewhat manual as compared to other approaches.
2. Use of JavaScript on client side is central to the discussed technique. Hence cross-browser portability can be an issue in situations where browser specific JavaScript features are used in an application.
3. As is common with any AJAX driven web application, page refresh and browser back button require special handling.
4. Also, benefits of the discussed technique are significant only when application use cases involve repeated activations of same views. That is, if a view is activated only once in a session then this approach doesn't provide significant benefit in comparison to the other mentioned approaches.

Table 5. Application implementation styles

Design Method	Description
Method-1	It made use of the proposed technique where markup and data were decoupled.
Method-2: Component based UI	It implemented the dynamic web page using ASP.NET web forms.
Method-3: Tag libs based UI	It implemented the same dynamic web page using ASP.NET MVC framework.

4.3 Experimentation Details and Results

To examine the effectiveness of the proposed technique, a reference implementation was developed for it. We compared the performance against: 1) an application which used a component based UI and 2) an application which used server-side tag library based UI. Functionality wise, the applications showed the users a single dynamic page which displayed a list of products entries similar to a shopping web site product catalogue. A product entry consisted of fields such as: Description summary (a hyperlink), Price, Availability status, User rating (a hyperlink) and a Thumbnail image (a hyperlink).

Each entry was put inside a grid cell on the HTML page. On one page the grid showed about 30-50 rows. This represents a fairly close scenario to most of the web applications that we studied in section-2. Table-5 shows the details about the implementation styles for each of the applications.

Each application was implemented using ASP.NET 2.0 framework on .NET 3.5. Web server was IIS 6.0 on Windows 7 (64bit) running on Intel Core2 Duo T6600 @ 2.2GHz processor with 4GB RAM. Each application was subjected to the load of 100-1500 users making 500 requests each.

Performance Indicator

The Performance Indicator (PI) can be chosen as the CPU usage, Memory usage and Throughput. In the present work, the performance of the proposed technique has been tested with respect to all the above mentioned three criteria. The PI is observed for each method-i and is then normalized with respect to the value of PI for the proposed technique. The normalized index for a PI-r is calculated as:

$$T_r = \frac{T_r^p}{T_r^i} \tag{1}$$

where, T_r^i is value of PI-r as observed with method-i, and T_r^p is the value of same indicator as observed with proposed approach. The results obtained from the experiments are shown in the Figure-3.

We found that with proposed approach the average throughput was about 7 times better than the other two approaches. The server-side memory usage was, on average, about 12% less than that of tag library based UI, and about

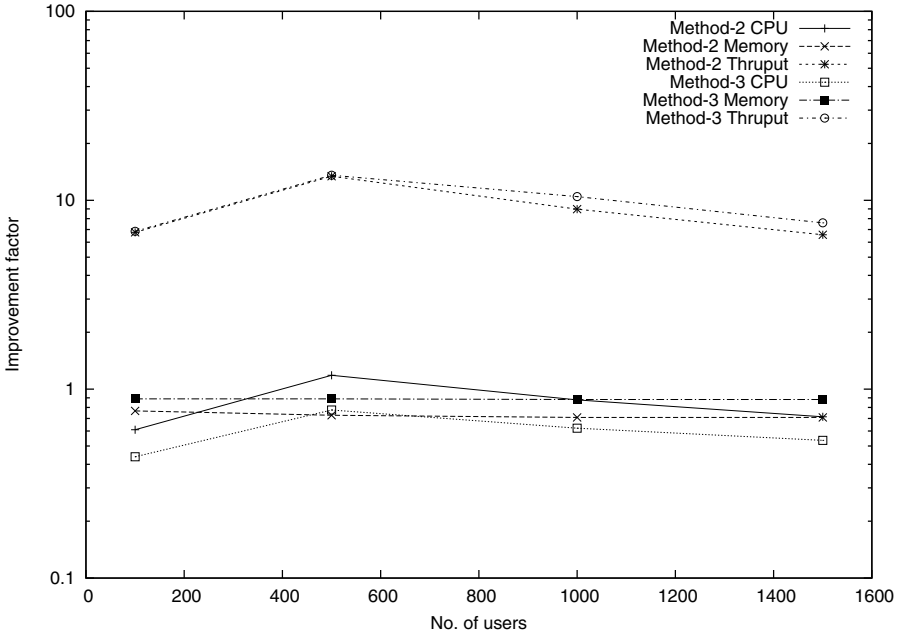

Fig. 3. Efficiency comparison

Table 6. Performance indicator values

Tag Libraries UI			**Components Based UI**			
CPU	Memory	Thruput	CPU	Memory	Thruput	**Users**
0.439	0.888	6.852	0.610	0.767	6.760	100
0.776	0.888	13.581	1.184	0.728	13.405	500
0.621	0.882	10.454	0.878	0.708	8.969	1000
0.536	0.882	7.591	0.714	0.708	6.562	1500

30-40% less than component based UI approach. CPU consumption was 25-65% less in comparison to tag libraries based UI, and was about 15-24% less than the component based UI approach.

The general trend with the variation of number of concurrent users load is as expected. All performance indicators first slightly improve before they finally plateau out. Initial small improvement with the increase in users load happens due to utilization of cached or one-time created objects. Once the cached objects are built up, then its contribution towards overall performance saturates and doesn't reflect with the further increase in user load.

HTTP requests load for web applications was generated via Apache JMeter [17] and the measurements were done via the following tools: Windows Process Explorer [18], YourKit profiler for .NET [19].

5 Conclusions

We systematically investigated how the *useful information* is spread and delivered via various structural elements of the HTML markup in dynamic web applications. It was observed that a major chunk of a served HTML page contains just the markup; the real useful information content is confined to only a small set of HTML tags and tag attributes. We exploited this finding to propose an efficient technique to deliver dynamic content where we decoupled the markup from the data so that they could travel between client and server independently and only on demand. Our test results show that the proposed technique reduces the use of server-side computing resources such as CPU, memory and network bandwidth.

For instance, we observed that on the www.youtube.com home page just the HTML tags account for about 37% of the HTML size – and we are considering all the tags content as *useful information* here, which in practice is not the case since, for example, the content of *SCRIPT* tag (which often doesn't change) itself is accounting for about 17% of the HTML size here. Average size of the www.youtube.com HTML is about 70000 bytes. So the potential saving per repeated pageview is about $0.37 \times 70000 = 25900$ bytes of bandwidth. According to Alexa statistics [20] www.youtube.com attracts about 25% internet users everyday. By taking into account the reported bounce rate (i.e. the percentage of visits to youtube.com that consist of a single pageview) of about 25%, and assuming that there are only about 1 billion Internet users measured by Alexa per day, simple math shows that youtube.com could potentially save at least about $25900 \times 0.25 \times (1 - 0.25) \times 1000000000 = 4856250000000$ bytes (i.e. 4522.735 GB) of bandwidth every day. And we are not even looking at the server-side CPU and memory savings at this scale.

The above example clearly shows the usefulness of our technique in heavily used dynamic web applications, particularly the ones deployed in cloud platforms where the resource usage is metered and billed at a very granular level. The other important possible applications of our technique are in low bandwidth clients and handheld devices.

References

1. Alexa Inc. Alexa - top sites by category: Shopping, `http://www.alexa.com/topsites/category/Top/Shopping` (retrieved: March 2011)
2. Alexa Inc. About the alexa traffic rankings, `http://www.alexa.com/help/traffic-learn-more` (retrieved: March 2011)
3. Lin, Q.Z.Z., Wu, J., Zhou, H.: Research on web applications using ajax new technologies. In: MMID 2008, pp. 139–142 (December 2008)
4. Paulson, L.D.: Building rich web applications with ajax. IEEE Computer 38(10), 14–17 (2005)
5. Json, D.C.: The fat-free alternative to xml (January 2011), `http://www.json.org/fatfree.html`

6. Google Inc. Google web toolkit developer guide (January 2011), http://code.google.com/webtoolkit/doc/1.6/DevGuide.html
7. Microsoft Inc. Microsoft silverlight reference documentation (January 2011), http://msdn.microsoft.com/en-us/library/cc838158%28VS.95%29.aspx
8. Apache Software Foundation. Apache struts developer guide (March 2010), http://struts.apache.org/2.1.8/docs/guides.html
9. Ruby on Rails Foundation. Ruby on rails developer guide (March 2010), http://guides.rubyonrails.org
10. Zend. Php zend framework manual (January 2011), http://framework.zend.com/manual/en/
11. Sun Microsystem. Java server faces (jsf) reference documentation (January 2011), http://java.sun.com/javaee/javaserverfaces/reference/docs/index.html
12. Microsoft Inc. Microsoft asp.net reference documentation (2011), http://msdn.microsoft.com/en-us/library/dd394709%28VS.100%29.aspx
13. Gross, C.: Ajax Patterns And Best Practices, ch.3. Dreamtech Press (2007)
14. JQuery. Jquery documentation, http://docs.jquery.com/Main_Page (retrieved: January 2011)
15. The Dojo Foundation. Dojo toolkit guide (March 2011), http://dojotoolkit.org/reference-guide/
16. Yahoo Inc. Yui library manual (March 2010), http://developer.yahoo.com/yui/
17. The Apache Software Foundation. Apache jmeter (March 2010), http://jakarta.apache.org/jmeter/index.html
18. Microsoft TechNet. Windows sysinternals - process explorer v12.03 (February 2011), http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx
19. YourKit LLC. Yourkit profiler 5 for .net (March 2010), http://www.yourkit.com/.net/profiler/index.jsp
20. Alexa Inc. Youtube.com site info., http://www.alexa.com/siteinfo/youtube.com (retrieved: March 2011)