Impact of Remote User Interface
Design and Delivery on Energy Demand

Tomas Cerny
Computer Science, FEE, Czech Technical University,
Charles Square 13, Prague 2, Czech Rep.
Email: tomas.cerny@fel.cvut.cz

Michael J. Donahoo
Computer Science, Baylor University,
Waco, TX, USA
Email: jeff_donahoo@baylor.edu

Abstract—Client-side User Interface (UI) for web applications
clearly plays a critical role in user performance and efficiency.
Growing user expectations drive UI design to greater function-
ality with ever increasing expectations for rich and continuous
interactivity. Such increases require greater and greater com-
putational resources. At the same time, web applications are
increasingly accessed through mobile, battery-powered devices,
such as notebooks, tablets, smartphones, and even watches.
In effect, users are simultaneously increasing dependence on
battery power and the pace of battery discharge with demanding
applications. While UI design often considers factors such as
usability, bandwidth consumption, etc., little consideration is
given to the impact rendering and delivery design have on
energy consumption. While we may expect novel technologies
to expand battery capacity, the demands consistently outpace
improvements. Careful consideration of UI design strategy may
reduce the energy demands placed to the users device. This paper
presents a study considering existing UI design and delivery
strategies and evaluates their impact on energy consumption.

Index Terms—energy impact; user interface; separation of concerns

I. INTRODUCTION

Existing UI research [1], [2], [3] usually addresses UI abilities,
personification [1], context-awareness or cross-platform compati-
bility [2]. Conventional web-based approaches target design and
maintenance simplification and tool-aided approaches [2], [4], [5], [6]. Research also often focuses on
server-side scalability and UI responsiveness, targeting fast
end-user delivery [1], [7], [8].

The growing market of battery-equipped mobile devices
demonstrates the need for energy-friendly UI design and
delivery approaches. While the forecast for battery capacities
anticipates growth reports12, the ever-increasing reliance on
mobility insures the demands on battery energy will con-
tinue to outpace supply. Considerable reduction in energy
consumption of user’s device could be achieved by choosing
an appropriate UI design, delivery, and rendering strategy by
reducing client-side computational resource demands.

This paper considers contemporary approaches of UI
design, delivery, and rendering of data presentations in web
applications. It investigates their impact on energy consumption
at the user’s device. We compare the conventional, server-side
UI design approach, represented by JavaServer Faces (JSF)
[9], with the approach brought by Google Web Toolkit (GWT)
[7] and AngularJS (AJS) [8]. The study is extended to consider
the impact of JSF PrimeFaces (PF) [10] library providing
better usability and attractive look and feel. Moreover, the
delivery impact is considered from the concern-separating
perspective. To provide a broad study, the Distributed Concern
Delivery (DCD) [1], [11] is compared with the conventional
delivery approach. The DCD extension to JSF, AJS and GWT
is considered and compared with the original.

The presentation is organized as follows. Section 2 intro-
duces web UI design approach, their specifics and abilities. A
case study is described in Section 3 evaluating various factors
to draw the energy impact. Section 4 concludes the paper.

II. BACKGROUND, DESIGN AND DELIVERY APPROACHES

Conventional web applications provide the client UI in the
form of HTML, supplemented with images, style sheets, Java
Script (JS), JSON, XML and other sources. The client-server
interaction uses the HTTP protocol built on top of the TCP/IP
protocol requiring an initial three-way handshake to establish a
connection and four-way handshake to terminate. HTTP brings
multiple simplifications and also transmission optimization.
For instance, it supports content compression to reduce the
volume. Next, seldom-changing resources can be cached by
clients to further improve the interaction. Furthermore, web
browsers open multiple simultaneous connections to server for
parallelization. To avoid handshake overhead, connections are
reused. HTTP works well for partial fragment requests usually
involving asynchronous server calls for web resources.

The server-side UI description of a web application may
use a dynamic interpreter considering the HTML description
extended with a special markup for dynamic behavior or con-
tent resolution with underlying application context, allowing
to bind its data values, use variables, conditionals, interaction,
etc. (e.g., PHP). Alternatively, the description uses an abstract
language defining the UI [6], and the result eventually trans-
forms to HTML or JS before leaving the server (e.g., JSF [9]).

Java Enterprise Edition uses JSF [9] for the UI, and thus
we use it as the reference UI implementation. In fact, no
matter the approach, the conventional UI design suggests to
describe a particular UI page combining components, layout,
data binding, validation rules, constraints, security, etc., so
that the page is self-descriptive. The most simplistic view

---

may provide the Composite design pattern [3]. JSF uses an abstract description on top of HTML introducing new components. It puts the main effort on the server-side where the JSF interpreter interprets the UI description and assembles a component tree that represents the UI. JSF renderer traverses the tree and derives HTML descriptions for the client delivery.

The approach of GWT [7] uses abstraction but on completely different level than JSF. JSF uses a domain-specific language that provides a binding mechanism to Java. GWT UI descriptions use Java and thus improve type safety. GWT Java descriptions get compiled, rather than interpreted, into a JS representation. The client loads the JS that gets interpreted at the client-side, minimizing server-side involvement. The compilation of Java description uses various optimization heuristics to minimize the JS volume and furthermore produces JS for various browsers. Large part of the JS is cacheable, but there are also some uncacheable fragments. The data values in GWT are usually requested through JSON as a separate piece of information. The nature of GWT fits to interactive pages. Its use for large, data-oriented systems might be demanding, considering the volume of produced JS. Since the UI logic loads with the UI, there is a potential for offline interaction. Both JSF and GWT introduce design abstraction classifiable as model-driven design [3], when seeing the JSF/Java description as model and HTML as the target.

The main argument of its rival, AJS [8], is that Java philosophy is too distant and not corresponding to web-development. The high abstraction brings difficulty for debugging low level optimization and inability to apply changes to generated JS. Similar to GWT, the AJS expect data to be provided as a separate piece of information. The difference is that it is low-level, involving JS development. GWT loads the page states all at once, while AJS suggests incremental state extension, that fits better with data-oriented systems. Next, there is a novelty in the client-side involvement and introduction of templating and data decoration. It allows defining templates used for data presentation. Thus each data instance displayed on the page follows the same template, reducing restatements and transmission size. The templating mechanism expects client’s browser to execute decoration demanding resources.

[1] argues that conventional design approach, although easy to comprehend, is not efficient for context-aware situations, since each new condition or context may lead to a design of a novel UI description. Instead, it look for inspiration in Aspect-Oriented Programming (AOP) and applies the approach to the UI design [3]. This separates out the descriptions of presentation components, layout, data binding, validation rules, constraints, security, etc. The UI assembly executes at runtime from these individual concerns. This avoids the inefficiency of conventional design when dealing with context-awareness by designing page per context.

From the delivery perspective, no matter the HTML/JS format, JSF/GWT/AJS provides a single, tangled UI description to clients. Additionally, GWT/AJS separate out the data values in JSON from descriptions. Even the AOP-based approach tangles all concerns at the server-side.

[1] promotes the AOP idea to delivery. Having the descriptions separated and untangled brings new perspective to the client-server interaction. Instead of providing the client the tangled HTML/JS description as a single block of information, it is possible to preserve the concern separation at the delivery level and introduce DCD [11]. DCD separates data presentation concerns, such as component selection and presentation, validation, structure, layout and data values. Such separation maintained at the client-side further extends client abilities. For instance, it extends the caching options to reuse components presentation and selection, validation, layout and structure concerns that would be tangled in the conventional approach. Similarly to conventional approaches, DCD provides the main HTML document with the page layout. Data presentations are assembled at the client-side from separately-requested concerns (JS/JSON). This leaves the decision of concern reuse and caching to the client-side. The context-awareness with DCD is less demanding with respect to transmission, since only changed concerns are provided, rather than all tangled concerns. Moreover, having the UI concerns separated brings finer granularity that further allows to classify the concerns to platform-specific and platform-independent and consequently simplify the design of platform-aware UIs [11].

DCD can be illustrated by the following example. Conventional design tangles all concerns together, introducing restatements and replication resulting in a single HTML document. DCD provides concerns to clients separately, which reduces the total volume and avoids restatements. After the initial page layout HTML document gets interpreted by the client, other concerns are requested concurrently. This increases the number of requests, while providing opportunity for parallel concern request processing. Thus the same UI with DCD may, in total, transmit less, in less time, while extending concern caching.

III. Case Study

For the purpose of UI design and delivery approach comparison, we conduct the following experiment. A sample UI page is designed using the JSF/PE/AJS/GWT. Furthermore, the DCD extension is also integrated to JSF/AJS/GWT to receive its impact. This gives 7 page prototypes. The impact of the approach is considered from both the client and server perspectives. When client’s web browser requests a page to load, we evaluate multiple criteria. Specifically, from the client-side perspective, we consider the page load time until all fields are rendered, used browser tab panel (tab) CPU sampled per 1ms and measured the total cores usage in ms, the tab’s allocated memory (Mem), the transmitted volume, total uncompressed size, total amount of requested resource, total packets both directions, total packet size and energy impact calculated by Mac OS X3. From the server-side, we consider CPU and Mem used for serving the client.

3http://www.teckrevue.com/tip/use-activity-monitor-energy-tab-os-x-mavericks: The number that is a relative measure of the energy impact of an app or process, taking into account factors such as overall CPU utilization, idle energy draw, and interrupts or timers that cause the CPU to wake up. Scale is 0 till indefinite high, while max number reported is 780. The lower the number, the less energy impact an app or process has.
The exact configuration is as follows: The UI page is based on the ACM-ICPC contest registration system. The fully functional page shows user profile information in an editable form with backend corresponding to the ACM-ICPC system. The page data presenting form has 22 input fields that consider input validation, various UI components, simple layout, and data binding. The considered UI frameworks versions are JSF.2.1.18, PF.4.0.7, AJS.1.4.0, and GWT.2.6.0. All images and style sheets are stripped out from the evaluation, leaving only the native JS libraries for the approach to operate. The application backend uses Java Enterprise Edition 6 on JBoss AS 6.2 server, running Java 8 with Postgres 9.3.4. The server-client connection has no bandwidth/latency restrictions, operating on localhost. The physical machine has a 2.3 GHz Quad-core Intel Core i7 with 16 GB Memory. The Google Chrome 44.0.2403.155 web browser used in the experiment in incognito mode with Task Manager and Developer Tools. The used monitoring tools are JConsole, Instruments 6.4, Activity Monitor 10.10.0 and Packet Peeper 2014-06-15.

Each page prototype is deployed at the same server and with criteria measurement repeated 5 times, while interleaving different prototypes one by one to minimize skew results due to possible Mac OS X internal tasks. The measurement considers two situations: Web browser with disabled/enabled caching. The DCD approach gives us the possibility to cache structural information in case no context changes to the UI are made, which is the more common case [11]. To provide broad evaluation and impact for context-aware situations, we also consider the situation when structural information change and plot both situations for DCD evaluation.

Table I shows results for the cache-disabled evaluation. Next, we discuss the outcome. JSF represents the standard approach and is compared with other approaches. The expectation of extension PF is improved usability and look and feel. Naturally, we expect that nearly all measured criteria get worse, which is also apparent from the results with mostly increased transmission volume and processing factors. PF does not really bring any alternative approach consideration; on the other hand, it gives us assurance that the measured values reflect the expectation when compared to JSF. The situation is more interesting considering AJS/GWT that brings significant resource utilization twist. See the Browser tab CPU and Mem\(^4\) and compare it with Server CPU and Mem utilization. The rendering is offloaded to the client. The transmission increase is caused by JS libraries. Since most of them are generic, they do not affect the later cache-enabled evaluation. Both AJS/GWT have an extra JSON request for data values, and the increased volume corresponds to packets. The tab measured energy impact corresponds to the resource utilization in the UI assembly, almost doubling the impact for GWT.

The DCD extension significantly impacts JSF. Consider the load time as well the transmission or uncompressed size. It further decreases the server CPU and Mem utilization; on the other hand, the energy impact is similar, since the tab CPU/Mem does not significantly change as it assembles the UI from concerns. Furthermore, DCD has more requests and involved packets. The DCD impact on AJS/GWT is for most factors marginally negative, which is caused by the AJS/GWT nature that already delegates resource utilization to the client-side. The energy impact is negative to AJS since the larger volume is processed and the UI rendering applies DCD assembly as well as AJS assembly. DCD energy impact to GWT gets slightly better. This may correspond to the CPU usage peak revealed in the evaluation. The plain GWT demands considerable CPU in a short time for the initial resource processing producing high CPU peak; DCD flatten the CPU peak for page processing, while using the same CPU resources. Even though the overall DCD effect seems counterproductive for AJS/GWT, it may affect caching abilities that we reveal and the later the cache-enables evaluation.

Fig. 1 gives the relative percentage impact of a particular approach (not considering PF JSF extension) and given criteria when compared to the JSF approach. As an example, consider the first criteria showing JSF DCD improving the page load time by 33% and AJS extending it by 7%.

The cache-enabled results are expected to improve most of the measured criteria. Table II shows the impact on approaches without DCD. The load time improves considerably, even though the tab CPU does not change as significantly and tab

### Table I: Criteria Results for Uncached Measurement

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Unit</th>
<th>PF</th>
<th>JSF</th>
<th>AJS</th>
<th>GWT</th>
<th>JSF DCD</th>
<th>AJS DCD</th>
<th>GWT DCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page load time</td>
<td>ms</td>
<td>381</td>
<td>255.9</td>
<td>275</td>
<td>280.4</td>
<td>170.4</td>
<td>282</td>
<td>281.2</td>
</tr>
<tr>
<td>Browser tab CPU</td>
<td>MB</td>
<td>321.2</td>
<td>177.2</td>
<td>311.3</td>
<td>349.8</td>
<td>174.4</td>
<td>310.3</td>
<td>349.4</td>
</tr>
<tr>
<td>Browser tab Mem</td>
<td>MB</td>
<td>76</td>
<td>60.7</td>
<td>72.1</td>
<td>74</td>
<td>59.8</td>
<td>72.2</td>
<td>74.9</td>
</tr>
<tr>
<td>Uncompressed size</td>
<td>KB</td>
<td>675</td>
<td>80.5</td>
<td>164</td>
<td>177</td>
<td>51.8</td>
<td>181</td>
<td>178</td>
</tr>
<tr>
<td>Compressed trans. KB</td>
<td>170</td>
<td>19.7</td>
<td>56.5</td>
<td>60.6</td>
<td>15</td>
<td>62</td>
<td>60.5</td>
<td></td>
</tr>
<tr>
<td>Resources</td>
<td>KB</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Packets</td>
<td>KB</td>
<td>178.4</td>
<td>24.8</td>
<td>63.6</td>
<td>68.3</td>
<td>22</td>
<td>71.2</td>
<td>69.3</td>
</tr>
<tr>
<td>Page CPU MB</td>
<td>271.2</td>
<td>171.8</td>
<td>64.2</td>
<td>32.8</td>
<td>96.2</td>
<td>61.6</td>
<td>35.3</td>
<td></td>
</tr>
<tr>
<td>Server Mem MB</td>
<td>18.6</td>
<td>18.2</td>
<td>8.5</td>
<td>3.8</td>
<td>7.6</td>
<td>9.6</td>
<td>4.2</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Relative criteria percentage impact based to the JSF approach.

---

311 textual, 4 select menus, 3 dates, 3 radio options and one checkbox.

---

4It is important to point out that, while the Browser tab memory use indicate 60-76 MB, it includes the allocation for the tab itself with 55.4 MB, although we preserve the total number to draw the practical impact.
the advantages, at the same time there are more efforts in resource allocation at the server-side. Extension brought by DCD positively balances the traditional design with a delivery approach that reduces server-side involvement. An important factor of the evaluation centered around the volume of transmitted, compressed information and transmission improvements with caching. While caching positively impacts page load times, surprisingly it does not significantly reduce the energy demands, since the overall uncompressed volume of information is utilized by web browsers. The DCD ability to cache structural information brings reductions to server-side resource utilization for all considered approaches. No matter the browser caching or delivery approach the clients-side resource utilization for all considered approaches. No caching structural information brings reductions to server-side resource utilization for all considered approaches.

Future work involves direct analysis on iOS for energy evaluation and impact of the network throttling. Furthermore, optimization to energy consumption will be addressed.

ACKNOWLEDGMENT

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/201/OKH3/3T/13

REFERENCES


