Cluster Comput (2016) 19:1853–1863
DOI 10.1007/s10586-016-0665-7

On energy impact of web user interface approaches

Tomas Cerny[1] · **Michael Jeff Donahoo**[2]

Received: 20 June 2016 / Accepted: 5 October 2016 / Published online: 15 October 2016

Abstract Developers base selection of a User Interface (UI) development approach on functionality, development and maintenance costs, usability, responsiveness, etc. User expectations continue to grow for greater functionality and continuous interactivity, extending demands on computational resources. To facility scale, recent approaches push more UI computation to clients. Such client-side delegation of functionality increase, continuous usage, and localized computation create ever-growing energy demands, which may negatively impact battery life on mobile platforms. Nonetheless, developers given little attention to the power demands aspects of UI framework selection. We evaluate the impact of contemporary UI framework selection on resource utilization and energy consumption. We suggest an alternative delivery approach designed to preserve low energy demands on clients while still allowing offloading of computation from server to client. Our work focuses on web-based mobile applications; however, we believe our approach to energy demand reduction and framework evaluation to be generally applicable.

Keywords Energy impact · User interface design · Separation of concerns · Resource distribution · Web applications

✉ Tomas Cerny
tomas.cerny@fel.cvut.cz

Michael Jeff Donahoo
jeff_donahoo@baylor.edu

1 Department of Computer Science, FEE, Czech Technical University, Charles Square 13, 12135 Prague 2, Czech Republic

2 Department of Computer Science, Baylor University, Waco, TX 76798, USA

1 Introduction

Researchers and industry experts endeavor to design User Interface (UI) frameworks for web applications that bring attractive UIs to users. What are the intended properties that such a UI should have? There are many attributes to consider. For instance, researchers in Human–Computer Interaction (HCI) aim to provide a UI with efficient interaction and usability that adapts to each individual user [18,20,21]. At the same time, Software Engineers aim to provide high responsiveness to user actions with emphasize on availability, performance, scalability [8] or testability [2]. Moreover, to allow heterogeneous devices and native applications, researches aim to provide platform-independent representations of the UI [7,21]. All these properties focus solely on user application experience.

Some UI properties impact areas much less directly visible than attractive UIs. In order to support adaptive UI features or personalization, developers may spend a tremendous amount of work [5] implementing a variety of context-specific versions of UI. Many performance optimizations can be applied to provide good responsiveness and high scalability, while investing into a large infrastructure. Over time many frameworks emerged aiming to minimize development costs ranging from those that aim to minimizes development efforts, reduce code volume [5] and others offloading computations to clients [12] involving JavaScript (JS).

To highlight the complexity of UI development, consider the overall application development and the portion devoted to UI. Approximately 48 % of application code and 50 % of development time is devoted to implementing UIs [15]. When further considering support of multiple platforms and context-awareness, the complexity exacerbates [8].

To make UI development and maintenance inexpensive and systems scalable, researches search for high-level and Domain Specific Languages (DSL) to describe UIs [14,19,24] for Model-Based Approaches (MBA) [1] or ways to offload computations to clients [12]. The motive is clear, but the result may possess deficiencies. For instance, high-level DSL and MBA approaches transform the UI description into low-level code involving HTML, JS, JSON and CSS. The transformation usually produces output tangling various UI concerns that negatively influences the volume of information transferred between server and clients. Even worse [5], the concern tangling might be apparent even in the high-level UI description. The server becomes responsible to deal with tangling and to process and provide a large amount of information. Nevertheless Google Web Toolkit (GWT) noticed the above deficiency and utilized the transformation of the high-level UI description into JS in order to perform most of the computation at the client-side. From a high-level perspective, the transformation produces various combinations of interaction states and provides them to clients at once, so that clients perform computations locally and provides the results to the server in a lite format, such as JSON.

Let us summarize the introduced options. One option is to let the server to build and render the UI at the server-side and then provide it to clients in rather voluminous HTML and supplementary resources. When client changes the state, the server-side rebuilds the UI (or its fragment) and sends the updated version to a particular client. The other option provides clients the instruments to render the UI locally with its various states. Server then only provides a lite amount of information, usually application data that the client manages.

We can see that the second approach separates data from the other resources used to render the UI. The first approach involves server-side resources, while clients only process the received HTML with supplementary resources. The second approach relocates computational resource use to the client-side. Thus the client is involved in UI rendering process, while server provides limited amount of information and reduces resource usage.

Clearly, the second approach reduces server resource usage as well as lowering the volume of information being send from server to clients. However when clients possess limited resources, the first approach seems favourable.

Battery capacity of mobile devices is a limiting resource that might be given higher value when choosing UI design. When we look at the growing market of battery-equipped mobile devices, the forecast for battery capacities anticipates growth reports.[1,2] The ever-increasing reliance on mobility insures the demands on battery energy will continue to outpace supply [6]. Thus, when considering battery capacity limiting, we should take into account the energy impact of UI design and delivery approaches. An appropriate approach can reduce the energy consumption related to UI rendering.

This paper considers contemporary approaches of UI designs, delivery, and rendering of data presentations in web applications. It investigates their impact on resource usage and energy consumption at the user's device. In a case study, we compare the conventional, server-side UI design approach, represented by the standard Java Enterprise Edition technology JavaServer Faces (JSF) [3] with the client-side UI design approaches brought by Google Web Toolkit (GWT) [12] and AngularJS (AJS) [10]. The study is extended to consider the impact of a library [25] providing better usability and attractive look and feel for the server-side approach. Furthermore, the evaluation considers caching abilities of the above approaches and its impact on energy demands.

While it is expected that the client-side approaches demand more energy from clients upon UI rendering and reduce the server-side resources, does the delivery strategy impact the resource usage? Next, we aim to answer the question whether the above approaches change the resource usage characteristics with an alternative delivery approach. Our results show that an alternative delivery strategy has significant impact on the server-side approaches. While it preserves low energy demands for client-side UI rendering, it considerably reduces the server-side resource usage.

The paper is organized as follows. Section 2 introduces conventional web UI design approaches. Section 3 provides related work on power consumption. A case study comparing conventional approaches is given in Sect. 4 evaluating various factors to draw the energy impact. Section 5 describes an alternative delivery approach. The alternative approach is evaluated in a study in Sect. 6. Validity threats of the studies are provided in Sect. 7. Finally, the last section concludes the paper.

2 Background, design and delivery approaches

Conventional web applications provide their UI to clients in the HTML format, usually supplemented with media files, CSS, JS, JSON, XML and other sources. The client-server interaction uses the HTTP(S) 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 [23]. HTTP brings multiple transmission optimizations, i.e., it supports content compression to reduce the volume; seldom-changing resources can be cached by clients to further improve the interaction. Next, web browsers enable multiple simultaneous connections to the server for parallel resource requests. Moreover, to mitigate the handshake overheads, connections are reused for multiple requests. In addition, HTTP works well for partial fragment requests usu-

[1] http://news.mit.edu/2015/yolks-and-shells-improve-rechargeable-batteries-0805.

[2] http://news.mit.edu/2015/solid-state-rechargeable-batteries-safer-longer-lasting-0817.

ally involving asynchronous server calls for web resources, i.e., Asynchronous JS and XML (AJAX).

Servers provide UIs to web clients in HTML. The particular UI at the server-side can be described through HTML extended with a special markup or use an abstract language defining the UI. The special markup adds dynamic behavior and content resolution with the underlying application context, allowing to bind its data values, use variables, conditionals, interaction, etc. (e.g., PHP). The second option that uses an abstract language [14] aims to simplify the description with domain-specific constructs. The abstract description eventually transforms to HTML or JS before leaving the server (e.g., JSF [3], GWT [12]). The abstract description has the advantage of possible optimization or context-specific transformation, i.e., producing web-browser specific versions. There also exist alternative approaches, such as model-based [26], generative [24] or inspection based [7].

In this work, we compare conventional UI design approaches. Java Enterprise Edition comprises one of our candidates, which suggests its standard technology JSF [3] to describe UI. Such approaches, represented by JSF, describe a particular UI page combining components, a layout, integrating data bindings, validation rules, constraints, security, etc., so that the page is self-descriptive. The most simplistic view of this approach involves the composite design pattern [11]. In addition, JSF uses an abstract description with XML-like constructs, providing various widgets and components. This approach puts the main effort on the server-side. The JSF interpreter interprets the UI description and assembles a component tree that represents the UI. To derive the HTML description for client delivery, a renderer traverses the component tree and produces the HTML outcome.

Our next approach involves UI rendering clients. This approach is utilized, for instance, in GWT [12]. It uses abstraction but on a completely different level than JSF. While JSF uses a domain-specific language that provides a binding mechanism to Java, GWT uses Java to describe the UI. This may improve type safety; on the other hand it is questionable whether Java, a general-purpose language, fits well with designing web UIs. The principal advantage of GWT is that Java descriptions are compiled, rather than interpreted. The compilation of Java description uses various optimization heuristics to minimize the JS volume. The product is a JS representation for various web-browsers. In the life cycle, clients load the entire JS representation and interpret it locally at the client-side, minimizing server-side involvement and reducing its resources. A large part of the JS is cacheable, although certain fragments are uncacheable. The client-side interpreter requests the data values from the server through JSON as a separate resource. This approach separates data values from the rest of the UI. The nature of GWT fits well to interactive pages, i.e. email clients, interactive consoles, etc. Since the UI logic loads with the UI, there is a potential for offline interaction. Limiting is the use on devices with limited resources, such as cellphones and smart watch. Furthermore, it does not fit to large, data-oriented systems with multiple independent pages. The produced JS might be demanding with respect to volume or the complexity of processing. At the same time, both JSF and GWT introduce design abstraction classifiable as model-based [5] where the JSF/Java description provides the model and the HTML/JS product comprises the target.

The high-level of abstraction introduced in JSF/GWT brings difficulty for debugging and low-level optimization as well as an inability to apply changes to generated JS. AngularJS (AJS) [10] also suggests that the Java philosophy is too distant and does not correspond well to web UI design. AJS is more low-level sort of development, involving JS code. A similarity to GWT is that AJS expects data to be provided as a separate piece of information. The difference is that GWT loads the page states all at once, while AJS suggests incremental state extension. The incremental state approach fits better with data-oriented systems. Next, AJS brings a novelty to the client-side involvement; it introduces a templating mechanism and data decoration. This allows defining templates used for data presentation, and thus each data instance displayed at a particular page follows the same template. The advantage brought by the templates is a reduction of restatements on a page as well as decreased transmission size. At the same time, the templating mechanism expects a client's browser to execute decoration, demanding additional resources.

Next, we may consider the three approaches from the perspective of resource involvement and energy demands. JSF is an approach that involves server-side rendering, and thus most of the resources are involved at the server. The processing tangles various concerns together, such as data, presentations, security, layouts, validations, etc. The tangled product of the server-side is HTML. Clients receive the produced HTML and interpret it without considering the individual concerns. From the interaction perspective, clients request HTML, and other requested resources, i.e., JS, media and CSS. The HTML content has rather larger volume and may contain repetitions introduced by concern tangling. For instance, consider that two text fields contain the same component declaration and different binding. There are two component descriptions in the HTML since the reuse is not possible due to the tangling with bindings or other concerns such as layout. Thus the server-side utilizes resources for rendering and transmission involves large volumes. Although the compression reduces the negatives, both server and client interact with the larger volume.

The approach used by GWT compiles the Java description before application deploy producing outcome in JS format representing a variety of applications states. The product divides on two parts, cacheable and uncacheable fragments. Furthermore the data are requested through a lightweight JSON format. Regarding interaction, first time visiting clients load small HTML and consequently request large cacheable JS with page logic and small volume uncacheable JS fragment, JSON, and all these extended with other resources. A client renders a particular UI based on context locally. Returning visitors only load small portion of information, specifically the uncacheable JS and particular data in JSON. Most of the UI rendering involves client-side resources.

Finally, AJS provides an alternative involving both sides. The server is responsible for defining layouts, presentation definitions, security, pages logic, etc. The client is responsible to render data in provided templates. Clients in addition to UI description also receive templates in HTML used to display data received through a separate resource in JSON format. This mitigates restatements and repeated information. Next, it allows processing certain amount of rendering at the server side, while rendering data elements such as forms, tables, and reports locally by clients. In comparison to GWT, only a limited page state is transferred upon request. From the interaction perspective, clients load HTML, JS, JSON and other resources.

3 Related work on power consumption

Various related work [9,17,22] considers power consumption of servers to reduce involved costs. Intel [22] suggests that servers on average consume 250 Watts (W). While cooling is an important factor in data centers, it is not the main consumer of power. As suggested by [22], processors and memory consume most of the power, followed by power supply loss. Disk drives are not significant in terms of power consumption. Processor power consumption varies from 45–200 W. Power drain grows with processor utilization. The authors suggest an estimation of power consumption (P) at any specific processor utilization (n). It can be calculated if the power consumption is known for maximum performance (P_{max}) and idle utilization.

$$P_n = (P_{max} - P_{idle}) \times \frac{n}{100} + P_{idle}. \tag{1}$$

Through empirical measurement of various servers using a power meter, this approximation has verified to be accurate $\pm$ 5 % across all processor utilization rates [22].

The next, largest power consumer is memory. The consumption differs when idle or active (50 % bandwidth and 67 % read with 33 % write). The average use is 5–12 W when active; the idle use is 2–5 W per module. The power drain grows with capacity and frequency. For instance, 16 GB 1600 MHZ DDR3[3] consumes 15 W on idle and 21–23 W under load.

In [9] the authors consider energy consumption modelling for data centers. They present distribution of power usage by components. They suggest the distribution across components to CPU 33 %, DRAM 30 %, disks 10 %, network 5 % and the rest by other sources. In their model, they consider complex parameters to derive the energy use involving the sum of energy consumed by CPU, memory, disk, and network interface card.

Since many energy consumption models show complexity, [17] suggest a simplified model. Their model involves dynamic and static fragments of power consumption of the CPU. Next, they consider cache access and DRAM power consumption involving access misses. To simplify the model, they consider multiple constants received from empirical measurements. In their benchmarks, the CPU power drain is proportional to the frequency and number of cores. Specifically they measure approximately 70 W for 2.39 GHz, single core and 95 W for quad core.

An extended study on power consumption for mobile phones is conducted [4]. This publication provides a detailed overview of energy usage and discusses the significance of the power drawn by various components. As expected the power drain grows with display backlight brightness; the range is 7.8 and 414 mW on the minimal and maximum backlight. A set of benchmarks is applied to see the impact on power drain from CPU, RAM, Flash storage, Network (WiFi, GPRS) and GPS. Various scenarios are evaluated to see the usage of these elements. For web browsing, the backlight plays a dominant role, although its usage to 67 % reduces from the maximum 414 mW to less than 150 mW. With WiFi access, the GSM, WiFi and Graphics modules drain each less than 100 mW, the CPU, LCD modules as well as the rest (including RAM) drains each less than 50 mW. With the GPRS option, the GSM module drain grows slightly above 200 mW with the WiFi is disabled. The WiFi/GPRS consumption is up to 430.4/500 mW. The results were validated on Nexus One, HTC Dream and Openmoko Neo Freerunner phones with variations. Percentage power drain is as follows. Backlight is not considered and can range from 50 to 10 % from the following numbers. Graphics hardware uses 24 %, GSM uses 23 %, WiFi 16 %, CPU 14 %, LCD 11 % and the rest (including RAM) 12 %.

The energy model considered in the paper shows energy for each usage scenario as a function of time (t). Specifically, for web browsing (on WiFi) it is:

$$E_{web}(t) = (0.43\,W + P_{BL}) \times t, \tag{2}$$

[3] http://www.servethehome.com/testing-power-savings-low-voltage-135v-kingston-memory-intel-avoton/

where the P_{BL} is the backlight power in watts. Benchmarks involving CPU with the highest frequency demanded up to 900 mW on Nexus One and 500 mW on HTC phones.

Next, we provide an overview of the up-to-date devices. We consider various specifications and benchmarks.[4] The overall power consumption from the tests for MacBook[5] for the Pro late 2015 version show idle 2.8–8.1 W and under load 52.4–62.8 W. The non-pro version shows 1.7–6 W on idle with maximum brightness and wlan. Under the load it shows 29.3 W, which goes down to 18.5 W within a few minutes. Lenovo ThinkPad-13 report[6] shows overview over multiple notebook brands with average idle power drain 5.5–6.7 W and average load 30.9–34.6 (Acer/Asus/Lenovo brands). The display consumption[7] can be as low as 17.4 W. Similar reports[8] given to phones show various brands with idle avg. drain 0.9–4.1 W and under the load on avg. 2.4–5.5 W. Standalone devices have higher consumption.[9] The power drain for iMac, Mac Mini and Mac Pro when considering the most power efficient versions show power drain in the range of 40–119 W (idle-max) for iMac, 4–85 W (idle-max) for Mac Mini and 43–203 W (idle-max) or Mac Pro.

When considering the impact of power drain in the context of web design, rendering and delivery, most work suggests that CPU usage is an important parameter as well as the network over wireless connection. Memory is also influential parameter. Attributes such as screen backlight or LCD (if applicable) are not impacted by UI approach. Graphics hardware may get a slight impact since we render similar product. We consider the above parameters in a case study.

4 Case study: evaluation of conventional approaches

For the purpose of UI design and delivery approach comparison, we conduct the following experiment. A sample UI page is designed using the approach of JSF, AJS and GWT. Furthermore, we consider a JSF extension that improves usability. The same page is designed and evaluated using the PrimeFaces (PF) [25] library.

The page representative is as follows. The UI page is based on the ACM-ICPC contest registration system[10] and builds on its backend. The fully functional page shows a user profile in an editable form. The system represents a production-level application. The specifics of the UI page are that it contains person information accessible through a web form that contains 22 input fields.[11] The form is an interface thorough which users edit system data. The form further considers input validation, simple layout, data binding, and security rules all reflecting the ACM-ICPC system.

The particular applications use these specific frameworks versions: JSF 2.1.18, AJS 1.4.0, GWT 2.6.0. and PF 4.0.7 All page images and CSS 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 application server, running Java 8 with Postgres 9.3.4.

The server–client connection has no bandwidth/latency restrictions, operating on localhost. The physical machine is a MacBook Pro (late 2013) with an Intel(R) Core(TM) i7-4850 HQ CPU @ 2.30 GHz Quad-core (with an Intel Iris Pro integrated GPU) with 16 GB Memory 1600 MHz DDR3 and 6 MB L3 Cache. The specification of the processor[12] suggests typical consumption 38.19 W (thermal design power 47 W), reviews[13] suggest its power consumption higher, idle takes 7.6 W and under load on avg. 80.4 W (88.5 W max). Similar results are given by [13] showing on load under a benchmark power drain in the range of 40–80 W. This corresponds to usual usage for power consumption of other CPUs.[14]

The Google Chrome 44.0.2403.155 web browser is used in the experiment in incognito mode with the Task Manager and Developer Tools. The monitoring tools are JConsole, Mac Instruments 6.4 (allowing us to tap a particular process or even web browser tab), Activity Monitor 10.10.0 and Packet Peeper 2014-06-15.

The impact of the approach is considered from both the client and server perspectives. When client's web browser requests the page to load, we evaluate multiple criteria and factors. Specifically, from the client-side perspective, we consider the page load time (until the entire page renders) and CPU use of the process representing the tab panel (tab) of web browser. Specifically, with sampling at an interval of 1 ms, we monitor whether any of the CPU cores is used and count the total number of cores involved. The measured number represents the total number of CPU units (core used in 1 ms)

[4] http://www.notebookcheck.net, https://support.apple.com.

[5] http://www.notebookcheck.net/Apple-MacBook-Pro-Retina-13-Early-2015-Notebook-Review.139621.0.html, http://www.notebookcheck.net/Apple-MacBook-12-Early-2015-Notebook-Preview.142672.0.html.

[6] http://www.notebookcheck.net/Lenovo-ThinkPad-13-Ultrabook-Review.166559.0.html.

[7] https://www.apple.com/euro/environment/reports/docs/15inch_MacBookPro_wRetina_PER_July2014.pdf.

[8] http://www.notebookcheck.net/Microsoft-Lumia-650-Smartphone-Review.165363.0.html.

[9] https://support.apple.com/en-us/HT201918, https://support.apple.com/en-us/HT201897, https://support.apple.com/en-us/HT201796.

[10] http://icpc.baylor.edu.

[11] 11 text inputs, 4 select menus, 3 dates, 3 radio options, 1 checkbox.

[12] http://cpuboss.com/cpu/Intel-Core-i7-4850HQ.

[13] http://www.notebookcheck.net/Apple-MacBook-Pro-Retina-15-Late-2013-Notebook-Review.120330.0.html.

[14] http://www.tomshardware.co.uk/skylake-intel-core-i7-6700k-core-i5-6600k,review-33276-11.html.

Table 1 Results for uncached measurement

Criteria	Units	PF	JSF	AJS	GWT
Page load time	ms	381	255.9	275	280.4
Uncompressed size	KB	675	80.5	164	177
Compressed trans.	KB	170	19.7	56.5	60.6
Resources		5	3	4	5
Packets		98	54	77	81
Packets size	KB	178.4	24.8	63.6	68.3
Client CPU units		321.2	177.2	311.3	349.8
Client CPU time span	ms	413	275	458	398
Client Mem (w tab)	MB	76	60.7	72.1	74
Client Mem (w/o tab)	MB	20.6	5.3	16.7	18.6
MAC energy impact		6.5	3.8	6.2	7.1
Derived energy estimate	J	8.3	4.6	8	9
Server CPU units		271.2	171.8	64.2	32.8
Server CPU time span	ms	186.5	144.4	213	245.25
Server Mem	MB	18.6	18.2	8.5	3.8
Aggregated CPU units		592.4	349	375.5	382.6
Aggr. Mem (w/o tab)		39.2	23.5	25.2	22.4

involved in page rendering and the tab overhead (the decimal point is introduced by averaging over multiple samples). For instance, 80 CPU units represents 8 cores used for 10 ms or 1 core used for 80 ms. Moreover it can represent 8 cores used for 5, 50 ms pause and again 5 ms with 8 cores the progress is usually with one peek upon the first request. We also provide the time between the first and last CPU unit use in the process. This time indicates the activity of the tab and spans over the page load time. Next, we consider the tab allocated memory (Mem) in MB. The tab itself allocates 55.4 MB. We also consider the transmitted volume in KB, as well as the total uncompressed size of the delivered content (KB). Moreover we consider the total amount of requested files/resources from the server and the total packets both directions with its total packet size (exceeds the compressed size, including the packet headers and overhead).

Mac OS X uses an energy model that is a measure of the energy impact of an app or a process. The model[15] takes into account the CPU utilization, idle energy draw, and interrupts or timers that cause the CPU to wake up. The scale is 0 to infinitely high, while max number reported is 780. The lower the number, the less energy impact an app or process has. The Mac energy impact is considered for the particular process.

From the server-side perspective, we consider the CPU used by the application server sampled by CPU units the same way as for the client, the CPU unit timing, and finally the Mem used for serving the client.

Each of the page prototypes is deployed at the same server and with criteria measurement repeated five times, while interleaving different prototypes one by one to minimize skew results. The measurement considers two situations, first time and returning visitor. We use web browser with disabled and enabled caching to emulate both situations (while the second has preloaded cache).

Table 1 shows results for the cache-disabled evaluation, and next we discuss the outcome. JSF represents the standard approach and is compared with other approaches. Naturally, we expect that nearly all measured criteria get worse for the usability extension PF, 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 comparing JSF to AJS/GWT that bring significant resource utilization twist between client and server. See the twist on client–server CPU units and Mem use. It is important to point out that, while the browser tab memory use indicates 60–76 MB, it includes the allocation for the tab itself with 55.4 MB (we show both values with and without the tab overhead). In comparison with server-side CPU units and Mem utilization, the rendering is offloaded to the client. The transmission increase is caused by JS libraries. This could change with the number of third party libraries. Since it could be seen as a threat to validity we can consider the later cache-enabled scenario that caches the JS libraries and limits the transmission. Both AJS/GWT have an extra JSON request for data values, and the increased volume also corresponds to involved packets. The tab measured energy impact for client corresponds to the resource utiliza-

[15] http://www.tekrevue.com/tip/use-activity-monitor-energy-tab-os-x-mavericks.

Table 2 Results for cached measurement

Criteria	Units	PF	JSF	AJS	GWT
Page load time	ms	284	218	251	257
Uncompressed size	KB	34.1	19.9	20.6	8.7
Compressed trans.	KB	4.6	3.3	4	4.2
Resources		1	1	2	3
Packets		51	51	48	66
Packets size	KB	8.2	6.8	8.1	9.2
Client CPU units		303	172	292	354
Client CPU time span	ms	369	235	372	378
Client Mem (w tab)	MB	78	60.3	72.5	74.4
Client Mem (w/o tab)	MB	22.6	4.9	17.1	19
MAC energy impact		5.8	3.6	6.1	6.6
Derived energy estimate	J	7.8	4.4	7.4	9.1
Server CPU unit		178	83.6	31.25	20.4
Server CPU time span	ms	128	89.4	195.8	225.8
Server Mem	MB	16.1	16.5	6.5	3.9
Aggregated CPU units		488	255.6	323.25	374.4
Aggr. Mem (w/o tab)		38.7	21.4	23.6	22.9

tion in the UI rendering, almost doubling when compared to JSF.

Moreover we consider a rather unusual perspective involving the aggregation of the CPU units of both sides. This represents the total number of CPU units needed to request, derive, response and render the UI. The same we consider for memory use.

In the measurement, JSF indicates the lowest demands on CPU and Mem to clients and thus smallest energy demands. AJS and GWT show a significant reduction of resource use for the server, but almost double demands for resources on the client side. The aggregated perspective gives the lowest CPU usage to JSF; the Mem involvement has small difference indicating the lowest demands for GWT.

Next, while not considering the screen backlight impact, or influence of other components, we aim to derive the energy consumption from the CPU usage. When we consider the CPU power demands 7.6 W in idle and 80.4 W under the maximum load, we can apply Eq. 1 (while originally demanded for servers). Since the power demands show linear growth, we consider the CPU units distributed equally in time using single core (out of 4). Below we install the values to the Eq. 1 multiplied by time in seconds to obtain Energy (Watt seconds - Joules).

$$E = \left((80.4 - 7.6) \times \frac{25}{100} + 7.6\right) \times t = 25.8 \times t(J) \quad (3)$$

When we consider the above equation and apply the client CPU units, we receive an energy estimate for the particular approach. This slightly corresponds to the Mac energy impact. JSF requires 4.6 J from clients' device to render the page, while AJS and GWT demands grow to 8 and 9 J. To give a practical example a 40-Watt Edison bulb would light 1 s for the same amount of energy that is approximately used to eight times render the JSF UI page in our study, while GWT version would only render four times.

The cache-enabled results are expected to improve most of the measured criteria due reduced resource requests and caching. Table 2 shows the impact on considered approaches. The load time improves considerably, even though the unexpected outcome is that the tab CPU units does not change significantly and tab Mem even grows. Transmission and uncompressed sizes drop considerably. Reduction is also apparent for the amount of resources and packets. Unexpectedly, the energy impact drops marginally, which corresponds to CPU and Mem. The positive impact is that the server CPU units reduce the involvement to almost half. Since the CPU units stay almost unchanged on the client-side we see that the energy demands are not impacted by caching. The caching impacts page load times and reduces the server-side involvement.

The main outcome from this study is that caching does not impact the client-side energy demands, which confirms the CPU usage, Mac energy impact and derived energy estimate. Caching does not impact the client-side memory usage. On the other hand caching is important since it improves page load times and reduces demands on server. The outcome shows that client-side rendering is more efficient than server resource utilization. The impact on clients is notable since JSF UI rendering demand almost half the energy of GWT. Furthermore, from the perspective of optimizing the combined CPU usage on both client and server sides, JSF shows to be the most efficient out of the considered approaches.

The actual energy impact to other devices is relevant to their power consumption under the idle and maximum load. In case the CPU usage would be the same, other computers such as Lenovo ThinkPad-13 with dual-core would have slightly smaller demands, even though the number of cores increases the left side of the Eq. 1. For the Lenovo report on other similar computers, the P would be approximately in the range of 18.2–20.7 W and consume 3.2–3.7 J for uncached JSF page rendering and 6.4–7.2 J for uncached GWT page.

5 Alternative delivery approach

We now consider whether the delivery approach impacts the resource usage and influences the energy demands. At this point, all the approaches deliver most of the UI concerns at once. AJS and GWT separate out data values and provide them as a separate JSON resource. Cerny et al. [8] argues that conventional design approaches, although easy to comprehend, are not efficient in terms of distributions of different concerns. A particular UI concern is, for instance, field presentation, layout, security, validation, data binding, and so on. While we usually think of each concern independent of others, when designing UI we tangle them together to describe a particular situation. The actual tangling disables particular concern reuse and causes complex design, repetitions and inefficiencies. In fact, this is a common issue for most programming languages that do not effectively address concerns that tend to cross-cut each other [16]. UI concerns are the types of concerns that are cross-cutting [5]. There are multiple approaches to deal with cross-cutting concerns effectively, but they require additional instruments to tangle the concerns upon execution. For instance, Aspect-Oriented Programming (AOP) [16] suggests to separate concerns from the base components, indicate the join points in components, and let an aspect-weaver decide how and which concerns tangle together to serve a particular request in given context.

The same sort of issue with concern tangling apparent in source code can be seen in delivery. Cerny et al. [8] suggests that it is possible to distribute concerns separated through different channels through individual requests. For instance, the server can provide a main HTML file that points to JS presentation and layout templates. These templates can be provided as part of a JS library and used by a client weaver. Such a weaver then requests page-specific data values from the server, similar as to GWT and AJS approaches. Furthermore, to derive the proper presentation, it requests meta-information of given data. Such meta-information provides the structural details, constraints, validation and security of data that are displayed on the page to properly derive the date presentation, i.e., form. While the same information is given to clients as in conventional approaches, it is divided to particular parts.

The client is then responsible to derive the UI data presentations locally. Since there are multiple resources and browsers support concurrency, they can be requested simultaneously and support page load. Each of the resources has a distinct life-span, and this allows us to cache and reuse the resources independent of each other. For instance, layouts and presentation templates do not change over the time, but when using a contemporary approach, there is only single source of information , which must be resubmitted. With the Distributed Concern Delivery (DCD) [8] approach, the selection of whether to cache a particular concern or not is possible. Moreover, there is no repetition in the provided resources as there would be, for example, in conventional JSF that declares the use of 11 text fields that mostly contain the same information. Finally, if such distribution exists, it is possible to look at the UI data definitions from the perspective of platform independence [7], and design native client applications reusing the server provided platform-independent concerns given in machine readable format.

Next, we implement and apply the DCD extension to JSF, AJS and GWT application and evaluate the impact of the delivery approach on resource utilization on the same case study.

6 Case study: evaluation of delivery approaches

The same configuration is applied for the DCD experiments, while the conventional approach results are apparent in Tables 1 and 2. The DCD approach gives us the possibility to cache more concerns, such as presentation and layout templates for the entire application. Furthermore, it is possible to cache the meta-information that provides UI structure for data. On the other hand, if the page changes based on the context, the meta-information can change, i.e., conditional field rendering, security changes, time-awareness, etc. To provide a broad evaluation and impact for context-aware situations, we consider the situation without caching of the meta-information. As an extension, we also show the situation for context-insensitive UIs.

Table 3 shows results for the cache-disabled evaluation with DCD. The DCD extension for the uncached scenario does not show any major impact to the client-side. We may notice reduced transmission for JSF and as expected more requests. The most interesting perspective is the server-side for JSF. The total number of CPU units and Mem drops almost in half and brings positive impact. On the other hand most other criteria, especially in AJS and GWT, are almost unchanged.

The cache-enabled results are expected to improve most of the measured criteria. Table 4 shows improvement in CPU

Table 3 Results for uncached measurement with DCD applied

Criteria	Units	JSF	AJS	GWT
Page load time	ms	170.4	282	281.2
Uncompressed size	KB	51.8	181	178
Compressed trans.	KB	15	62	60.5
Resources		5	6	6
Packets		64	89	87
Packets size	KB	22	71.2	69.3
Client CPU units		174.4	310.3	349.4
Client CPU time span		231.8	448.6	385.3
Client Mem (w tab)	MB	59.8	72.2	74.9
Client Mem (w/o tab)	MB	4.4	16.8	19.5
MAC energy impact		3.6	6.6	6.8
Derived energy est.	J	4.5	8	9
Server CPU units		96.2	61.6	35.3
Server CPU time span	ms	99	166	233
Server Mem	MB	7.6	9.6	4.2
Aggregated CPU units		270.6	371.9	384.7
Aggr. Mem (w/o tab)	MB	12	26.4	23.7

demands on client-side for JSF. The memory usage slightly drops for JSF as well. The transmitted sizes reduce, and mostly the uncompressed size reduces due to supported concern reuse and reduced repetitions. The energy demands on the client-side slightly reduce for JSF. There is not much positive impact to GWT and AJS from the client's perspective. The server-side perspective shows significant reduction for JSF regarding the involvement of both CPU and Mem. There is a slight improvement to server-side resources for AJS and GWT with the context-unaware version of DCD that caches the meta-information.

The DCD impact is positive for the server-side approach JSF. It significantly reduces the server-side involvement due to reduced rendering on the server-side. Furthermore, it has a slightly positive impact on clients-side. There is a significant impact to the page load times for JSF. DCD does not significantly impact the considered client-side approaches.

7 Validity threats to the case studies

7.1 Internal validity

To mitigate sudden impact of background processes, the measured results are averaged over five samples of each scenario interleaving the various scenarios. All the measurements took place the same day after computer full restart with no processes that would put high demands on resource, even though that our measurement involved a specific process, which further eliminates the background computer tasks. The web browser ran in incognito mode to avoid negative impact introduced by plugins. Certain elements measured in our study are not sensitive to CPU, such as transmission sizes, etc. The constellation of the studies considers a single computer, which reduces the risk of skewing results by network fluctuation.

Table 4 Results for cached measurement with DCD

Criteria	Unit	Context-aware			Context-unaware		
		JSF	AJS	GWT	JSF	AJS	GWT
Page load time	ms	166	255.2	255	155.2	240.2	237
Uncompressed size	KB	7.5	9.5	13.7	2.4	4.2	8.7
Compressed trans.	KB	3.5	3.4	5.5	2.1	2.1	4.2
Resources		3	3	4	2	2	3
Packets		56	56	70	48	57	70
Packets size	KB	8.8	8.7	11.1	6.2	6.7	10.1
Client CPU units		154	305	367	163	308	340
Client CPU time span		184.4	390.8	391.7	216.4	464	343.5
Client Mem (w tab)	MB	59.5	72.6	74.5	61.5	72	74.4
Client Mem (w/o tab)	MB	4.1	17.2	19.1	6.1	16.6	19
MAC energy impact		2.9	6.1	6.6	2.8	5.9	6.2
Derived energy est.	J	4	7.9	9.5	4.2	7.9	8.8
Server CPU units		37	43	25	29.8	29	14.2
Server CPU time span	ms	92	125.8	227	83.6	125.6	229.4
Server Mem	MB	5.6	6.9	4.1	5.2	6	3.4
Aggregated CPU units		191	348	392	192.8	337	354.2
Aggr. Mem (w/o tab)	MB	9.7	24.1	23.2	11.3	22.6	22.4

7.2 External validity

Our application, while one representative, corresponds to a real-world application. The selected page reflects part of the application. In the study, we aimed to mitigate the specificity of the particular pages and reduced the references to media files, CSS and third party JS, to obtain specifics and impact of a particular approach. While the caching scenarios reduce all cacheable resources from the transmission, they are still accessed in memory; thus consideration of other resources impacts memory usage. The considered applications are fully functional regarding data management. At the same time, the representative does reflect all aspects of UIs. It demonstrates a particular data presentation page. A variety of different pages may contain different volumes of data, distinct layouts and most likely reference many other resources from multiple destinations. The outcome is representative of a page with a form, usual for information systems or enterprise applications. The results cannot be considered for interactive consoles, report or large content pages.

Multiple other factors can play role in real-world application such as distance from a server, etc. Furthermore the results must be considered from the perspective that user does not reload the UI page all the time, but instead the page rendering happens occasionally, and user most of the time reads the content. We approximated the energy demands by an equation suggested for servers and involving solely the CPU usage not considering energy impact of other components. At the same time, the variance among particular applications correlated to the Mac energy impact.

This case study serves as a demonstration of particular approaches. Next, it demonstrates the impact of alternative delivery approach. The study considers the ability of a conventional web-browser, Chrome, although its alternatives provide similar results.

8 Conclusion

This paper considers contemporary UI design and delivery approaches from the perspective of resource utilization and energy impact. The outcome of the research shows, that there is always a trade off between server and client-side computation. While some approaches such as AJS and GWT target client involvement to positively impact server resource use and bring benefits to service providers, these approaches place high demands on energy consumption for clients. This can even double the energy demands for UI page rendering on client-sides when compared to server-side approaches. The traditional server-side approaches place lower energy demands on clients, while more significant efforts in resource allocation is apparent at the server-side.

Our proposed extension brought by DCD positively balances the traditional server-side UI design with a delivery approach that reduces the server-side involvement. In our study, DCD improved server-side involvement on a level competitive to client-side approaches. In many evaluations, we may see caching and its corresponding reduction of transmitted volume as important factors, positively impacting page load times. In our study, we show that these factors have only a small impact on CPU utilization and energy demands. Web browsers demand CPU to process the entire uncompressed volume, including the cached fragments.

Future work will involve similar analysis on a mobile platform. Next, a greater variety of pages will be evaluated to provide a suggestion on design flaws regarding to energy impact. Finally, we plan to implement a tool capable of energy evaluation for web pages, to provide immediate feedback on energy demands.

Acknowledgements This work was supported by the Grant Agency of the Czech Technical University in Prague, Grant No. SGS14/198/OHK3/3T/13.

References

1. Balme, L., Demeure, A., Barralon, N., Coutaz, J., Calvary, G.: Cameleon-rt: a software architecture reference model for distributed, migratable, and plastic user interfaces. Ambient Intell. **3295**, 291–302 (2004). http://www.springerlink.com/content/p27tmcvqw24r2ljj
2. Bures, M.: Framework for assessment of web application automated testability. In: Proceedings of the 2015 Research in Adaptive and Convergent Systems, pp. 512–514. ACM (2015)
3. Burns, E., Griffin, N.: JavaServer Faces 2.0, The Complete Reference, 1st edn. McGraw-Hill, Inc., New York (2010)
4. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pp. 21–21. USENIX Association, Berkeley, (2010). http://dl.acm.org/citation.cfm?id=1855840.1855861
5. Cerny, T., Cemus, K., Donahoo, M.J., Song, E.: Aspect-driven, data-reflective and context-aware user interfaces design. Appl. Comput. Rev. **13**(4), 53–65 (2013)
6. Cerny, T., Donahoo, M.J.: Impact of remote user interface design and delivery on energy demand. In: 2nd International Conference on Information Science and Security (ICISS), pp. 1–4. IEEE, Prague (2015)
7. Cerny, T., Donahoo, M.J.: On separation of platform-independent particles in user interfaces. Clust. Comput. **18**, 1215–1228 (2015). doi:10.1007/s10586-015-0471-7
8. Cerny, T., Macik, M., Donahoo, J., Janousek, J.: On distributed concern delivery in user interface design. Comput. Sci. Inf. Syst. **12**(2), 655–681 (2015). doi:10.2298/CSIS141202021C
9. Dayarathna, M., Wen, Y., Fan, R.: Data center energy consumption modeling: a survey. IEEE Commun. Surv. Tutor. **18**(1), 732–794 (2016). doi:10.1109/COMST.2015.2481183
10. Freeman, A.: Pro AngularJS, 1st edn. Apress, Berkely (2014)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co. Inc, Boston (1995)

12. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with the Google Web Toolkit. Manning Publications Co., Greenwich (2007)
13. Johnsson, B., Akenine-Möller, T.: Measuring per-frame energy consumption of real-time graphics applications. J. Comput. Graph. Tech. (JCGT) **3**(1), 60–73 (2014). http://jcgt.org/published/0003/01/03/
14. Karu, M.: A textual domain specific language for user interface modelling. In: Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering. Lecture Notes in Electrical Engineering, vol. 151, pp. 985–996. Springer, New York (2013). doi:10.1007/978-1-4614-3558-7_84
15. Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. In: Proceedings of the 2nd conference on Human System Interactions, HSI'09, pp. 225–232. IEEE Press, Piscataway (2009). http://portal.acm.org/citation.cfm?id=1689359.1689399
16. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: IECOOP'97-Object-Oriented Programming, 11th European Conference, vol. 1241, pp. 220–242. Springer, New York (1997)
17. Kim, M., Ju, Y., Chae, J., Park, M.: A simple model for estimating power consumption of a multicore server system. Int. J. Multimed. Ubiquitous Eng. **9**(2), 153–160 (2014)
18. Lehmann, G., Blumendorf, M., Albayrak, S.: Development of context-adaptive applications on the basis of runtime user interface models. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '10, pp. 309–314. ACM, New York (2010). doi:10.1145/1822018.1822068
19. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: a language supporting multi-path development of user interfaces engineering human computer interaction and interactive systems. In: Bastide, R., Palanque, P., Roth, J. (eds.) Engineering Human Computer Interaction and Interactive Systems, Lecture Notes in Computer Science, pp. 134–135. Springer, Berlin (2005). doi:10.1007/11431879_12
20. López-Jaquero, V., Montero, F., Real, F.: Designing user interface adaptation rules with t: Xml. In: Proceedings of the 14th International Conference on Intelligent User Interfaces, IUI '09, pp. 383–388. ACM, New York (2009). doi:10.1145/1502650.1502705
21. Macik, M., Cerny, T., Slavik, P.: Context-sensitive, cross-platform user interface generation. J. Multimodal User Interfaces **8**(2), 217–229 (2014). doi:10.1007/s12193-013-0141-0
22. Minas, L., Ellison, B.: The Problem of Power Consumption in Servers. Intel Press, Hillsboro (2009)
23. Mogul, J.C.: The case for persistent-connection http. SIGCOMM Comput. Commun. Rev. **25**(4), 299–313 (1995). doi:10.1145/217391.217465
24. Schlee, M., Vanderdonckt, J.: Generative programming of graphical user interfaces. In: Proceedings of the working conference on Advanced visual interfaces, AVI '04, pp. 403–406. ACM (2004). doi:10.1145/989863.989936
25. Varaksin, O., Caliskan, M.: PrimeFaces Cookbook. Packt Publishing, Birmingham (2013)
26. Wu, J.H., Shin, S.S., Chien, J.L., Chao, W.: An extended mda method for user interface modeling and transformation. In: Osterle H., Schelp J., Winter R. (eds) Fifteenth European Conference on Information Systems, IHsieh M-C (2007), pp. 1632–1642. (2007)

Tomas Cerny received his Bachelor's and Engineer's degrees from the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. Next, he received his M.S. degree from Baylor University and in 2016 finished Ph.D. in Information Science and Computer Engineering. Since 2009, he works as an Assistant Professor at FEE, CTU. His area of research is software engineering, separation of concerns, model-driven development, enterprise application development and networking.

Michael Jeff Donahoo received his B.S. and M.S. degrees from Baylor University and his Ph.D. in Computer Science at the Georgia Institute of Technology. He is currently an Associate Professor of Computer Science at Baylor University where he conducts research on networking, security, and enterprise application development.