Communicating Web Vessels: Improving the Responsiveness of Mobile Web Apps with Adaptive Redistribution

Kijin An and Eli Tilevich

Software Innovations Lab, Virginia Tech, USA
{ankijin,tilevich}@vt.edu

Abstract. In a mobile web app, a browser-based client communicates with a cloud-based server across the network. An app is statically divided into client and server functionalities, so the resulting division remains fixed at runtime. However, if such static division mismatches the current network conditions and the device's processing capacities, app responsiveness and energy efficiency can deteriorate rapidly. To address this problem, we present Communicating Web Vessels (CWV), an adaptive redistribution framework that improves the responsiveness of full-stack JavaScript mobile apps. Unlike standard computation offloading, in which client functionalities move to run on the server, CWV's redistribution is bidirectional. Without any preprocessing, CWV enables apps to move any functionality from the server to the client and vice versa at runtime, thus adapting to the ever-changing execution environment of the web. Having moved to the client, former server functionalities become regular local functions. By monitoring the network, CWV determines if a redistribution would improve app performance, and then analyzes, transforms, sandboxes, and moves functions and program state at runtime. An evaluation with third-party mobile web apps shows that CWV optimizes their performance for dissimilar network conditions and client devices. As compared to their original versions, CWV-powered web apps improve their performance (i.e., latency, energy consumption), particularly when executed over limited networks.

Keywords: Mobile Web Apps · JavaScript · Dynamic Adaptation · Program Analysis & Transformation · Web Frameworks

1 Introduction

Mobile web apps are fundamentally distributed: browser-based clients communicate with cloud-based servers over the available networks. Distribution assigns an app component to run either on the client or on the server. Some distribution strategies are predefined; for example, user interfaces must display on the client. Other distribution strategies aim at improving performance; for example, a powerful cloud-based server can execute some functionality faster than can a mobile

device. Network communication significantly complicates the device/ cloud performance equation. For a client to execute a cloud-based functionality, it needs to pass parameters and receive results over the network. Transferring data across a network imposes latency and energy consumption costs. For low-latency, high-bandwidth networks, these costs are negligible. For limited networks, these costs can grow rapidly and unexpectedly. The overhead of network transfer can not only negate the performance benefits of remote cloud-based execution, but also strain the mobile device's energy budget. Operating over limited high-loss networks requires retransmission, which consumes additional battery power [19]. Hence, fixed distribution can hurt app responsiveness and energy efficiency.

Changing the locality of a software component can be non-trivial due to the differences in latency, concurrency, and failure modes between centralized and distributed executions [20]. Researchers and practitioners alike have thoroughly explored the task of rendering local components remote. *Cloud offloading* moves local functionalities to execute remotely in the cloud [8,2,17,21]. Nevertheless, standard offloading is *unidirectional*: it can only move a client functionality to run on a server. If mobile web apps are to flexibly adapt to the ever-changing changing execution environment of the web, client and server functionalities may need to adaptively switch places at runtime.

We address this problem by adaptively redistributing the client and server functionalities of already distributed applications to optimize their performance and energy efficiency. Our approach works with full-stack JavaScript apps, written entirely (i.e., client and server) in JavaScript. By dynamically instrumenting and monitoring app execution, our approach detects when network conditions deteriorate. In response, it moves the JavaScript code, program state, and SQL statements of a remote service to the client, so the service can be invoked as a regular local function. To prevent cross-site scripting (XSS) or SQL injection attacks, the moved code is sandboxed, creating a separate context with reduced privileges for safe execution in the mobile browser. Thus, the same functionality can be invoked locally or remotely as determined by the current execution environment. To the best of our knowledge, our approach is the first one to support *bidirectional dynamic redistribution of distributed mobile web apps*. Moreover, to take advantage of our approach, a mobile app needs not be written against any specific API or be pre-processed prior to execution.

We called the reference implementation of our approach—Communicating Web Vessels (CWV)—due to its reminiscence of *communicating vessels*, a physical phenomenon of connected vessels with dissimilar volumes of liquid reaching an equilibrium. CWV balances mobile execution by adaptively redistributing functionalities between the server and the client, thus optimizing app performance for the current execution environment. Our contribution is three-fold:

1. A novel bidirectional redistribution approach that dynamically adapts distributed mobile apps for the current execution environment.
2. A reference implementation of our approach, CWV, that works with increasingly popular full-stack JavaScript mobile apps. Requiring no pre-processing,

CWV dynamically adapts apps by redistributing their JavaScript code, program state, and SQL statements at runtime.

3. A comprehensive evaluation with 23 remote services of 8 real-world apps. To assess the effectiveness of CWV's adaptations, we report on their impact on execution latency and energy consumption.

The rest of this paper is structured as follows. Section 2 motivates and explains our approach. Section 3 describes the reference implementation of our approach. Section 4 presents our evaluation results. Section 5 compares our approach to the related state of the art. Section 6 presents concluding remarks.

2 Approach

We first present a motivating example, then give an overview of CWV, and finally discuss our performance model.

2.1 Motivating Example

Consider *Bookworm*[1], an e-reader app for reading books on mobile devices. The app also provides text analysis features that report various statistical facts about the read books. The app is distributed: the client hosts the user interface; the server hosts a repository of available books and a collection of text processing routines. The current architecture of *Bookworm* is well-optimized for a typical deployment environment: a resource-constrained mobile device and a powerful server, connected to each other over a reliable network. For limited networks, the performance equation can change drastically. Hence, to exhibit the best performance for all combinations of client and server devices and network connections, the app would have to be distributed in a variety of versions. Even if developers were willing to expend a high programming effort to produce and maintain all these versions, network conditions can change rapidly while the app is in operation, necessitating a different client/server decomposition. Clearly, achieving optimal performance under these conditions would require dynamic adaptation.

Our framework, CWV, can adapt *Bookworm*, so its remote text processing routines could migrate to the client at runtime for execution. CWV monitors the network conditions, migrating server-side functions to the client and reverting the execution back to the server, as determined by the network conditions. The app can start executing with all the text processing routines running on the server. Once the network connection deteriorates, a portion of these routines would be transferred over the network to the client, so they could execute locally. CWV's static and dynamic analyses determine the dependencies across server functions and their individual computational footprints. This information parameterizes CWV's performance model, which determines which part of server functionality needs to migrate to the client under the current network conditions.

[1] http://bookworm-data-insights.herokuapp.com

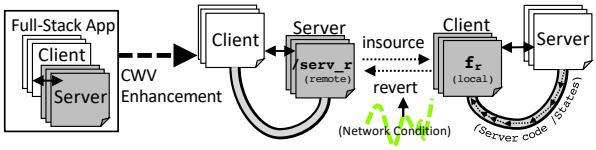


Fig. 1: Conceptual View of Communicating Web Vessels (CWV)

2.2 Approach: Communicating Web Vessels

To optimize the performance of mobile web apps for the current network conditions, Communicating Web Vessels (CWV) continuously apply these two operations (Fig. 1):

- $f_r = insource(/service\_r)$: The client requests that the server transfer the remote functionality($/service\_r$)'s partition f_r to the client.
- $revert(f_r)$: The client stops locally invoking the insourced partition f_r, and starts remotely invoking its original server version $/service\_r$.

2.3 Reasoning about Responsiveness

Responsiveness is a subjective criteria: application is responsive if the user perceives the time taken to execute app functionalities as "short". For this reason, we define the responsiveness of a remote execution as the total execution time that elapses between the client invoking a remote functionality and the results presented to the user. We define the response time of a remote functionality f_r as $RT(f_r)$. The $RT(f_r)$ mainly depends on the "server speed" and "network speed" parameters. We simplify the responsiveness of f_r by means of the execution time f_r on the server $T_{server}(f_r)$ and the remaining remote execution overheads. The resulting Round Trip Time (RTT) is highly affected by the current network conditions. To estimate the network conditions, CWV utilizes the RTT^{net} metrics, detailed in Section 3.3.

$$RT(f_r) = \begin{cases} T_{server}(f_r) + RTT^{net} & \text{remote exec.,} \\ T_{client}(f_r) & \text{local exec.} \end{cases} \quad (1)$$

If f_r is executed locally, the responsiveness becomes the execution time f_r on the client $T_{client}(f_r)$.

3 Reference Implementation

To move a server-side functionality to the client at runtime, one has to migrate both the relevant source code and program state, which has to be captured and restored at the client. JavaScript has a powerful facility, the `eval` function, which executes a JavaScript program passed to it as a string argument. One could simply duplicate the entire server-side code and its state, passing them to a

client-side `eval`. However, such a naïve approach would incur unacceptably high performance and security costs. Hence, our approach applies advanced program analysis and automated transformation techniques to minimize the amount of code to be transferred to and executed by the client (Sections 3.1 and 3.2). Furthermore, our approach establishes an efficient protocol for the transformed app to switch between different execution modes (Section 3.3), transferring the relevant code correctly and safely (Sections 3.4 and 3.5).

3.1 Analyzing Full-Stack JavaScript App

Server code comprises business logic and middleware libraries. The server-side business logic can include database access routines. The portion that needs to be insourced is business logic only. In other words, business logic must be reliably separated from all middleware-related functionality. To that end, CWV identifies the entry and exit statements of the business logic portion and then extracts all the code executed between these statements, converting that code to a new regular JavaScript function. All the dependent code of this new function is also extracted and transferred, thus producing a self-sufficient execution unit.

The specific steps are as follows. First, CWV normalizes the server code to facilitate the process of separating its business logic from middleware functionality. Then, CWV locates the statements that "unmarshal" the *client parameters* and "marshal" the *result* of executing the business logic. CWV automatically identifies these statements by capturing the client server HTTP traffic and instrumenting code at the server and at the client (Fig. 2-(a)). To that end, CWV uses Jalangi [16], a state-of-the-art dynamic analyzer for JavaScript. CWV modifies the built-in Jalangi's callback API calls to be able to detect the events that correspond to the "unmarshal/marshal" statements. By following these steps, CWV identifies the specific lines of code and variables that correspond to the entry and exit points of remote invocations, both at the server and the client.

The statements executed between these points comprise the server-side business logic and its dependent program states that may need to be moved to the client at runtime. To identify a subset of statements that satisfies a pair of entry/exit statements, CWV follows a strategy similar to that of other declarative program analysis frameworks that analyze JavaScript code by means of a datalog engine [18]. CWV encodes the declarative facts that specify the behavior of JavaScript statements of server program: 1) declarations of variables/functions, 2) their read/writes operations, and 3) control flow graphs. The dependency analysis query constructs a dependency graph between statements. Then, CWV solves constraints describing these points with the z3 engine [3] and then extracts them into a CWV-specific object that is movable between vessels (Fig. 2-(b)).

Some server-side program statements use third-party APIs, whose libraries and frameworks are deployed only at the server. CWV provides domain-specific handling of the statements that interact with relational databases. In particular, some statements interacting with a server-side relational database cannot be directly migrated to the client. As a specific example, consider the statement

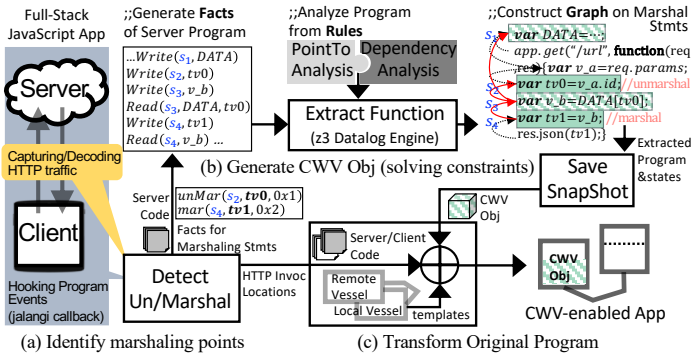


Fig. 2: Automated Program Transformation for enabling CWV

`mysql_server.query(SQL_STATEMENT)`, which queries the server-side MySQL database engine. Mobile clients can also use relational databases, but of a different type, a browser-hosted SQL engine. Hence, the database-related statement above should be replaced with `a_mobile_engine(SQL_STATEMENT)`. To identify such database-related statements, CWV instruments all function invocations whose arguments are SQL commands by using callback API of Jalangi. Despite the fragility of relying on the usage of SQL commands, our approach presents a practical solution for supporting domain-specific server-to-client migrations.

Finally, CWV transforms the identified entry/exit points at the client and server sides to insert the CWV functionality with the local and remote vessels respectively that we explain in the next section (Fig. 2-(c)).

3.2 Transforming Programs to Enable CWV

CWV enhances application source code to enable its transformation as follows.

Client Enhancements CWV transforms the identified HTTP invocation in the client program to be able to CWV's functionality as follow. The CWV-enabled client can operate and switch between these two modes: *Original* and *Local*. In *Original* mode, the app operates the original remote execution and can switch to Local mode by means of *Insourcing*. The *Local* mode designates that the local version of the insourced remote functions is to be invoked and can revert to the original mode by means of *Reverting* (See Fig. 3). To switch to a mode, the client invokes `fuzzMode(mode)` that simply fuzzes a certain parameter of the HTTP command that invokes the original remote service name. For instance, the client can dynamically fuzz a remote service `"/a_service"` (Original Request) into `"/a_service?CWVmode=Local"` (Local). And the app initiates the movement of the relevant remote server code and execution states *rcwv* to the client by fuzzing the original invocation into `"/a_service?CWVmode=Insourcing"` (Insourcing Request).

Insourcing CWV moves a set of received server statements into a client's container, referred to as the *local vessel*. Initially, the local vessel is empty. When the client device determines to switch from the *Original* mode into the *Local*

mode, the app issues the Insourcing Request and then invokes the `moveToLocalVessel(rcwv)` call, only then adding received server code and state to the local vessel. The client and server share all the referenced names for global entries added to the local vessels. To that end, CWV also adds a special-purpose global object for the client, *lcwv*. CWV defines the object's properties, initially contain nothing. This object is used for storing functions and other JavaScript objects received from the server[2]. Finally, the app fuzzes the HTTP command into Local `"CWVmode=Local"` to change the current mode. After that, invoking the `rebalance()` function compares the local replica's execution time with that of its original remote version.

Reverting If the local execution stops being advantageous, the app with *Local* mode reverts to *Original* mode and clears the local vessel with `clearLocalVessel()`, overriding the local vessel into the empty function again. And then, the app switches the mode by fuzzing HTTP command into the original mode.

Server Enhancements In a CWV-enabled app, the server part can operate in one of three modes to respond the client's requests: *Original*, *Insourcing*, and *Local*. With the detected entry/exit points of a remote functionality, CWV transforms it to be able to detect the mode switching queries and switch to the client-requested modes. The *Original* mode refers to the original unmodified execution, with the exception for the profiling of the time taken to execute the program statements that implement business logic $T_{server}(f_r)$ of the Equation (1). The client uses resulting performance profiles to ascertain the current network conditions RTT^{net} from the measured response time $RT(f_r)$. And $T_{server}(f_r)$ will be used to determine a threshold when to switch modes.

In the *Insourcing* mode, the server responds to the client's special insourcing query by serializing the relevant portions of a given remote functionality into a JSON string. To that end, CWV calls $saveSnapshot(f_r)$, whose invocation creates a snapshot of the remote functionality f_r. CWV adds to the server part a special-purpose global object, *rcwv*, which represents a *remote vessel*. This object's properties contain the extracted functions, $rcwv.main, rcwv.ftns[0], \cdots, rcwv.ftns[k]$ and their corresponding saved states for global variables $rcwv.gvars[0], \cdots, rcwv.gvars[l]$. To migrate f_r with database dependent statements, CWV takes a snapshot of database's table in terms of SQL commands to enable restoration in the client $rcwv.sql[0], \cdots, rcwv.sql[m]$. To implement $saveSnapshot(f_r)$, CWV instruments 1) the declarations of global variables and 2) *Call Expressions* of embedded SQL statements extracted by the constraints solving phrase. Finally, in the *Local* mode, the server executes no business logic, but responds to periodic pings from the client. Based on the roundtrip time of these pings, the client monitors the network conditions to detect if the *Local* mode execution no longer provides any performance advantages and then switches the app to the *Original* mode.

[2] The properties of *lcwv* are the same as of the remote object *rcwv*

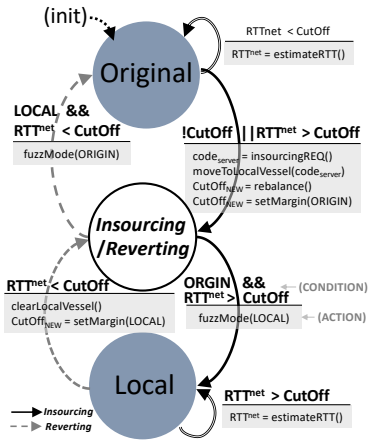


Fig. 3: Insourcing/Reverting to transition between modes in CWV-enabled Client

Input: raw network delay RTT^{net}_{raw}, current mode $m^{(k)}$ and current cutoff $\tau^{NET(k)}_{cutoff}$
Output: next cutoff $\tau^{NET(k+1)}_{cutoff}$ with a margin and next mode $m^{(k+1)}$ for CWV-enabled Client

```
//Remove spike by adaptive Kalman Filter
RTT^net ← estimateRTT(RTT^net_raw);
if RTT^net_filtered > τ^NET(k)_cutoff && m^(k) == Origin then
    //Profiling the difference for execs T̄ : rebalance
    τ^NET(k+1)_cutoff ← T̄_server(f_r) − T̄_client(f_r);
    //Set margin to the next cutoff condition
    margin ← (1 − θ) · RTT^net_filtered;
    τ^NET(k+1)_cutoff ← min(τ^NET(k+1)_cutoff, margin);
    m^(k+1) ← Local;
end
if RTT^net_filtered < τ^NET(k)_cutoff && m^(k) == Local then
    //Set margin to the next cutoff condition
    margin ← (1 + θ) · RTT^net_filtered;
    τ^NET(k+1)_cutoff ← max(τ^NET(k)_cutoff, margin);
    m^(k+1) ← Origin;
end
```

Algorithm 1: Updating Cutoffs and Modes

3.3 Updating Modes and Cutoff Latency

The transition diagram in Fig. 3 shows how an app can transition between different modes. CWV-enabled client always starts in the Original mode. An insourcing request issued in the Original mode can be either fulfilled (i.e., switching to the Local mode) or declined (i.e., continuing to execute remotely in the Original mode), with the latter incurring a large performance overhead. To avoid this overhead, the system determines the optimal time window for issuing "Insourcing Request" as soon as the app is automatically initialized with a couple of original executions. The procedure that determines the window is as follows. First, the client profiles both $RT(f_r)$ and $T_{server}(f_r)$ by means of multiple "Original Requests" during the initialization (Section 3.2). After that, the procedure invokes the "Insourcing Request" and extrapolates how much time it would take to execute the same business logic locally $T_{client}(f_r)$.

Estimating Network Delay CWV-enabled mobile clients continuously monitor the underlying network conditions. The client collects the RTT^{net}_{raw} metric that represents raw network delay. Specifically, the client is continuously monitoring the RTT^{net}_{raw} by subtracting $T(f_r)$ from $RT(f_r)$, which are obtained from the server. Since the raw roundtrip is subject to sudden spikes [7], CWV filters out such temporary fluctuations by applying an adaptive filter [11], which calculates the covariance matrices and noise values for RTT^{net}_{raw} and then estimates the RTT^{net} metric in Equation (1).

Cutoff network latency The resulting difference between the local and remote execution times is used as the threshold that determines when switching to the Local mode would become advantageous from the performance standpoint. In

other words, the difference value is compared with the overhead of network communication, and when the latter starts exceeding the former, the app switches to the Local mode. We define this network condition as *cutoff network latency*, τ_{cutoff}^{NET}. Thus, a CWV-enabled app obtains this threshold as soon as it start executing, and then stays in the *Original* mode until reaching the *cutoff*. Then, it tries switching to the Local mode. Because this request is executed only upon reaching the *cutoff*, it is more likely to be fulfilled as offering better performance.

Since switching between modes incurs communication and processing costs, frequent switching in response to insignificant network changes should be prevented. To that end, the *margin* parameter expresses by how much the network conditions need to change and remain changed. Algorithm 1 explains how the *margin* and the current cutoff latency $\tau_{cutoff}^{NET(k)}$ determine the next cutoff latency $\tau_{cutoff}^{NET(k+1)}$. The margin parameter θ prevents switching in response to insignificant $\tau_{cutoff}^{NET(k)}$ changes. After switching to the Local mode, the app periodically pings the network to determine if the current conditions are advantageous for reverting to the Original remote mode.

3.4 Synchronizing States

Some remote services can be invoked by means of HTTP POST, PUT, DELETE, which are all state-modifying operations. Invoking an insourced stateful remote service locally modifies its state, which must be synchronized with its original remote version via some consistency protocol.

Mobile apps are operated in volatile environments, in which mobile devices become temporarily disconnected from the cloud server. To accommodate such volatility, CWV's synchronization is based on a weak consistency model. As an implementation strategy, we take advantage of a proven weak consistency solution, Conflict-Free Replicated Data Types (CRDT), which provide a predefined data structure, whose replicas eventually synchronize their states, as the replicas are being accessed and modified. In CRDTs, the concurrent state updates can diverge temporarily to eventually converge into the same state, as long as the replicas manage to exchange their individual modification histories [6].

Specifically, CWV wraps the replicated 'database' and 'global variables' of *cwv* objects into the 'CRDT-Table', and 'CRDT-JSON' of CRDT templates[3], respectively. To keep track of changes and resolve conflicts, these CRDT-structures provide the API calls `getChanges` and `applyChanges`. By continuously applying/transmitting the reported changes, the device-based clients and the cloud-based server maintain their individual modification histories and exchange them, thus eventually converging to the same state. To that end, the cloud server periodically sends its state changes on *rcwv* to each client, while each client starts sending its state changes on *lcwv* to the cloud server, as soon as this client reverts to executing remotely.

[3] https://github.com/automerge/automerge

3.5 Sandboxing Insourced Code

Whenever code needs to be moved across hosts, the move can give rise to vulnerabilities unless special care is taken. The issue of insourcing JavaScript code from the server to the client is security sensitive. Server-side code has several privileges that cannot be provided by mobile browsers. In addition, as it is being transferred, the insourced code can be tempered with to inject attacks. Finally, the transferred segments of server-side database can be accessed by a malicious client-side actor. To mitigate these vulnerabilities, the insourced code is granted the least number of privileges required for it to carry out its functionality [15]. To that end, we *sandbox* the insourced code [12]. Specifically, CWV's sandboxing is applied to the entire local vessel. The insourced functionality has exactly one entry point through which it can be invoked. The sandbox guards the insourced execution from performing operations that require escalating privileges. Finally, because the insourced database data cannot be accessed directly, malicious parties would not be able to exfiltrate it. As a specific sandboxing mechanism, we take advantage of *iframe*, which has become a standard feature of modern browsers. An iframe creates a new nested browser context, separate from the global scope. Operating in a separate context precludes any shared state between the insourced code and the original client-based code. In addition, HTML5 supports the `sandbox` attribute to further restrict what iframes are allowed to execute[4]. It protects the client from the vulnerability related to client XSS. For instance, a sandboxed iframe is prohibited from accessing `window.localStorage[..]`.

4 Evaluation

Our evaluation seeks answers to the following questions:

- **RQ1:—Redistribution Adaptivity for different Devices:** How beneficial is CWV's redistribution for different mobile devices?
- **RQ2:—Redistribution Adaptivity for Networks:** How beneficial is CWV's redistribution for different networks?
- **RQ3:—Energy Savings:** How does CWV's redistribution affect the energy consumption of mobile devices?
- **RQ4:—Overheads:** When integrated with mobile apps, what is the impact of CWV on their performance?

4.1 Device Choice Impact

Dataset Our evaluation subjects are 23 remote services of 8 full-stack applications, 5 real-world full-stack mobile JavaScript applications, and 3 JavaScript distributed system benchmarks [21]. These subject apps use different middleware frameworks to implement their client/server (*tier-1/-2*) communication and database (*tier-3*), with these frameworks being most popular in the JavaScript ecosystem.

[4] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

To that end, we searched the results based on combinations of keywords for popular server and client HTTP middleware frameworks, curated by the community. For server-side keywords, we used 'Express', 'Koa', 'Restify', etc., while for client-side keywords we used 'Ajax', 'fetch', 'reactJS', 'Angular', etc. Table 1 summarizes their names and the number of source files; 4 subject applications contain database-dependent code. To answer **RQ1**, we tested how the introduced network delays affect different devices. At launch time for each device, CWV automatically calculates the *cutoff network latency* and applies it when scheduling mode switches to minimize the switching overhead. For example, CWV determined the cutoff network latency for the remote service "/hbone" as 26ms for device 1 (D1) in Table 1, having profiled the execution time at the server (T_{server}("/hbone")) and the client (T^{D1}_{client}("/hbone")) as 14ms and 40ms, respectively. Device 1 is a Qualcomm Snapdragon 616 (8 x 1.5GHz, Android), and Device 2 is an A8-iphone 6 (2 x 1.4GHz, iOS); Device 1 outperforms Device 2. The server is an Intel desktop (i7-7700 4 x 3.6 GHz, Ubuntu 16.04). We natively build the subject web apps (JavaScript, html, and CSS) for iOS and Android by using Apache Cordova, a cross-platform development framework. Table 1 demonstrates that the *cutoff latency* of Device 2 (τ^{D2}_{cutoff}) is always larger than that of Device 1 (τ^{D1}_{cutoff}).

Table 1: Subject Remote Services

Subject tiers,# of files	**Remote Services**	$\tau^{\mathbf{D1}}_{\mathbf{cutoff}}$ (msec)	$\tau^{\mathbf{D2}}_{\mathbf{cutoff}}$ (msec)
Bookworm (AngularJS, Express, *729 files*)	/ladypet	176ms	421ms
	/thedea	1120ms	2332ms
	/thered	158ms	424ms
	/thegift	97ms	120ms
	/bigtrip	146ms	224ms
	/offshore	619ms	1528ms
	/wallp	146ms	458ms
	/thecask	90ms	102ms
DonutShop (Ajax,Express,knex, *4.9k files*)	/Donut	0.66ms	1.54ms
	/Donut:id	0.71ms	2.2ms
	/Empls	0.55ms	1.33ms
	/Empls:id	0.81ms	1.23ms
recipebook (AngularJS, Express,MySQL,*8k files*)	/recipe	0.7ms	1.66ms
	/recipe:id	0.68ms	1.1ms
	/ingts/:id	0.82ms	2.3ms
	/dirs/:id	0.75ms	2.1ms
pstgr-sql (axios,restify, Postgres, *4k files)*	/user	1.33ms	2.71ms
	/user:id	1.72ms	2.92ms
chem-rules (fetch, koa, knex,*2.8k files*)	/hbone	26ms	59ms
	/molec	131ms	202ms
benchmark in [21] (Ajax, Express,*117 files*)			
str-fasta	/str-fasta	656ms	1424ms
fannk	/fannk	2576ms	4982ms
s-norm	/s-norm	1896ms	4873ms

4.2 Network Latency Impact

To answer **RQ2**, we set up a test-bed for evaluating network latency impact (See Fig. 5-(a)). Even though, network latency can be changed by controlling RSSI levels, we change network conditions explicitly by means of an application-level network emulator[5]. Then, we examine how CWV reacts by redistributing the running applications. In these experiments, the server and the mobile device are connected with a wireless router. We establish a high-speed wireless link between the router and the device (-55dBm or better). By configuring the router to different delays, we simulate different network conditions in the increasing order of delay. Our test-bed has a minimum delay of about 100ms for the simulator's

[5] https://github.com/h2non/toxy

zero delay. Therefore, our starting point is 100ms, with the delays increased in the increments of 20m, 50ms, and 100ms, based on the amount of *cutoff network latency* for each subject. For each increment, we measure the average delay in the execution of our subject applications (response time or responsiveness of a functionality), run in two configurations: (1) the original unmodified version (**Before**), (2) dynamically redistributed with CWV version (**CWV**). Fig. 4 shows the performance results.

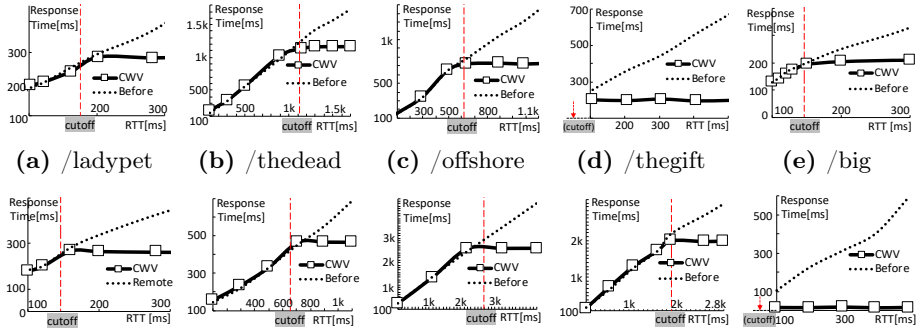


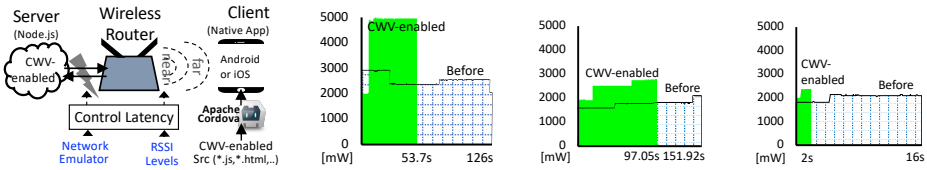
Fig. 4: Client's Responsiveness Comparisons. Cutoff equals to τ^{D1}_{cutoff} in Table 1.

Across all experimental subjects, the CWV-enabled configuration consistently outperforms the original version, once the network latency surpasses the *cutoff network latency* mark. Once the network delay reaches the *cutoff network*, the difference in performance starts increasing by a large margin, as accessing any remote functionality becomes prohibitively expensive. Before reaching the *cutoff network* mark, the majority of CWV-enabled apps and their original version exhibit comparable performance since two versions are operated in remote execution. When operating over a high-speed network, CWV-enabled apps remain in the original mode due to the remote execution's performance advantages. Some subjects consistently exhibit better performance when executed locally. These subjects with their relatively low utilization of server resources are better off not making any remote invocations, as the overhead of network delays is not offset by the server's superior processing capacity.

4.3 Energy Consumption

Next, we evaluate how much energy is consumed by a mobile device executing CWV-enabled and original versions of the same subjects (**RQ3**). We profile the energy consumption of Android devices with a Qualcomm's Trepn-Profiler. We executed each subject 100 times and collected the profiled results for battery power (mW). Fig. 5 shows the obtained samples of the battery power measurements over time. To test the consumed energy under a low speed network

environment, we placed the Android client device far from the wireless router, so the signal strength level(RSSI) was -75dBm. The resulting energy profiles in Fig. 5 show that CWV always uses more power than the original version despite shortening the execution time. Remote execution consumes no device power for executing the business logic, even if it takes much longer for the client to receive the results. By removing the need to communicate with the server, our approach shortens the overall execution time. Compared to the original version, our approach improves energy efficiency by as much as from **9.7J** to **74J** for a poor network condition. This result is not unexpected, as a large RTT causes longer idle periods between TCP windows [5]. Even tough, the device switches into the low power mode during the idle states, the longer total execution time consumes more energy overall.



(a) Testbed:Latency Control **(b)** /str-f (▼74J) **(c)** /thed (▼24J) **(d)** /hbon (▼31J)

Fig. 5: Testbed and Consumed Energy

4.4 Communication Overhead

To insource server execution, CWV serializes relevant code and state to transfer and reproduce at the client. To evaluate the resulting communicating overhead (**RQ4**), we compared the amount of network traffic during the regular remote execution for unmodified version (Tr_{reg}) vs. the additional traffic resulting from CWV insourcing server execution (Tr_{cwv}).

Among our subjects, the Bookworm app exhibits the largest of Tr_{orig}, as this app's remote services need to transfer not only the book content but also the statistical information extracted from that content. Whereas, the med-chem app shows the largest of Tr_{cwv}, as CWV needs to replicate all server-side DB entries. However, the transmitting overhead is occurred only once at initialization as these services are stateless. The resulting overall overhead ratio Tr_{cwv}/Tr_{reg} turned out to be **2.4** on average for our subjects (Fig. 6). To quantify the benefits of CWV's insourcing transferring only the necessary code and state, we also measured the overhead of the naïve approach, which transfers the entire server code and state to the client. The performance overhead of transferring everything is about two orders of magnitude slower than CWV, an unacceptable slowdown for any practical purposes (Fig. 6).

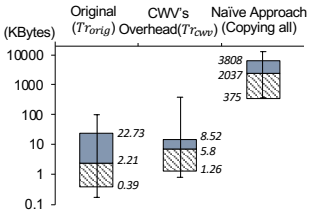


Fig. 6: CWV's Overhead

5 Related Work

Program Synthesis and Transformation: CWV automatically identifies a remote service's business functionality that satisfies the client's input and server's output constraints, akin to program synthesis systems, concerned with producing a program that satisfies a given set of input/output relationships. CodeCarbonReply [17] and Scalpel [2] supporting this functionality for C/C++ programs. The programmer manually annotates the code regions to integrate the transferred functionality. In contrast, CWV is both fully automated and dynamic, integrating program code and state at runtime.
Executing Code in a Mobile Browser: Ours is not the only approach that moves server-side components and data to the client. Meteor [14], a JavaScript framework, transparently replicates given parts of a server-side MongoDB at the client, so these parts can be used for offline operations. Browserify [1] enables a browser to use modulesin the same way as regular Node.js modules at the server. RT.js [4] prioritizes the execution of browser-based real-time jobs within the event queue, so they meet real-time constraints.
Adaptive Middleware: Several middleware-based approach has been proposed to reduce the costs of invoking remote functionalities. APE [13] defers remote invocations until some other apps switch the device's state to network activation. Similarly, to reduce the overhead of HTTP, HTTP requests in Android apps are automatically identified and then bundled into a single batched network transmission [10]. The e-ADAM [9] optimizes energy by changing various aspects of data transmission. CWV is yet another middleware, albeit tailored for the realities of adapting mobile apps by transferring program at runtime.

6 Conclusions and Future Work

This paper has presented Communicating Web Vessles (CWV), a dynamic adaptation approach that improves the responsiveness of mobile web apps under the ever-changing execution environment of the web. The CWV's reference implementation offers full automation and a low performance overhead. By featuring dynamic program analysis and transformation to ensure both correctness and efficiency, CWV adapts to dissimilar execution conditions by moving app functionalities from the server to the client and vice versa at runtime. As a future work direction, we plan to apply our approach to address the resource constraints and execution volatility of edge computing applications.

References

1. Ambler, T., Cloud, N.: Browserify. In: JavaScript Frameworks for Modern Web Dev, pp. 101–120. Springer (2015)
2. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 257–269. ISSTA 2015 (2015)

3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
4. Dietrich, C., Naumann, S., Thrift, R., Lohmann, D.: Rt.js: Practical real-time scheduling for web applications. In: 2019 Real-Time Systems Symposium (RTSS)
5. Ding, N., Wagner, D., Chen, X., Pathak, A., Hu, Y.C., Rice, A.: Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In: ACM SIGMETRICS Performance Evaluation Review. pp. 29–40 (2013)
6. Gomes, V.B., Kleppmann, M., Mulligan, D.P., Beresford, A.R.: Verifying strong eventual consistency in distributed systems. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–28 (2017)
7. Jacobsson, K., Hjalmarsson, H., Möller, N., Johansson, K.H.: Estimation of rtt and bandwidth for congestion control applications in communication networks. In: IEEE CDC, Paradise Island, Bahamas. IEEE (2004)
8. Kwon, Y.W., Tilevich, E.: Power-efficient and fault-tolerant distributed mobile execution. ICDCS '13, IEEE (2013)
9. Kwon, Y.W., Tilevich, E.: Configurable and adaptive middleware for energy-efficient distributed mobile computing. In: MobiCASE. pp. 106–115. IEEE (2014)
10. Li, D., Lyu, Y., Gui, J., Halfond, W.G.: Automated energy optimization of HTTP requests for mobile applications. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. pp. 249–260. IEEE (2016)
11. Marchthaler, R., Dingler, S.: Kalman-Filter, vol. 30. Springer (2017)
12. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Safe active content in sanitized javascript. Google, Inc., Tech. Rep (2008)
13. Nikzad, N., Chipara, O., Griswold, W.G.: APE: an annotation language and middleware for energy-efficient mobile application development. In: Proceedings of the 36th International Conference on Software Engineering. pp. 515–526. ACM (2014)
14. Robinson, J., Gray, A., Titarenco, D.: Getting started with meteor. In: Introducing Meteor, pp. 27–41. Springer (2015)
15. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63**(9), 1278–1308 (1975)
16. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 488–498 (2013)
17. Sidiroglou-Douskos, S., Lahtinen, E., Eden, A., Long, F., Rinard, M.: CodeCarbonCopy. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 95–105 (2017)
18. Sung, C., Kusano, M., Sinha, N., Wang, C.: Static DOM event dependency analysis for testing web applications. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 447–459 (2016)
19. Tsaoussidis, V., Badr, H., Ge, X., Pentikousis, K.: Energy/throughput tradeoffs of tcp error control strategies. In: Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications. pp. 106–112. IEEE (2000)
20. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: International Workshop on Mobile Object Systems. pp. 49–64. Springer (1996)
21. Wang, X., Liu, X., Zhang, Y., Huang, G.: Migration and execution of javascript applications between mobile devices and cloud. In: Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. pp. 83–84 (2012)