The Client Insourcing Refactoring to Facilitate the Re-engineering of Web-Based Applications

Kijin An

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Eli Tilevich, Virginia Tech, Chair
Godmar Back, Virginia Tech
Walter Binder, University of Lugano
Xun Jian, Virginia Tech
Francisco Servant, Virginia Tech

April 28, 2021
Blacksburg, Virginia

Keywords: Software Engineering, Reengineering, Web Apps, JavaScript, Mobile Apps, Program Analysis & Transformation, Middleware, Cloud Computing, Edge Computing

The Client Insourcing Refactoring to Facilitate the Re-engineering of Web-Based Applications

Kijin An

(ABSTRACT)

Developers often need to re-engineer distributed applications to address changes in requirements, made only after deployment. Much of the complexity of inspecting and evolving distributed applications lies in their distributed nature, while the majority of mature program analysis and transformation tools works only with centralized software. Inspired by business process re-engineering, in which remote operations can be insourced back in house to restructure and outsource anew, this dissertation brings an analogous approach to the re-engineering of distributed applications. Our approach introduces a novel automatic refactoring—Client Insourcing—that creates a semantically equivalent centralized version of a distributed application. This centralized version is then inspected, modified, and redistributed to meet new requirements. This dissertation demonstrates the utility of Client Insourcing in helping meet the changed requirements in performance, reliability, and security. We implemented Client Insourcing in the important domain of full-stack JavaScript applications, in which both the client and server parts are written in JavaScript, and applied our implementation to re-engineer mobile web applications. Client Insourcing reduces the complexity of inspecting and evolving distributed applications, thereby facilitating their re-engineering. This dissertation is based on 4 conference papers and 2 doctoral symposium papers, presented at ICWE 2019, SANER 2020, WWW 2020, and ICWE 2021.

The Client Insourcing Refactoring to Facilitate the Re-engineering of Web-Based Applications

Kijin An

(GENERAL AUDIENCE ABSTRACT)

Modern web applications are distributed across a browser-based client and a remote server. Software developers need to optimize the performance of web applications as well as correct and modify their functionality. However, the vast majority of mature development tools, used for optimizing, correcting, and modifying applications work only with non-distributed software, written to run on a single machine. To facilitate the maintenance and evolution of web applications, this dissertation research contributes new automated software transformation techniques. These contributions can be incorporated into the design of software development tools, thereby advancing the engineering of web applications.

Dedication

To my family:

my wife Mihee Lee,

my son Aiden Sooha An

Acknowledgments

I would like to express my deepest appreciation to my advisor Eli Tilevich[1]. Whenever I had difficulties in my PhD, he provided me with unwavering support and guidance, otherwise none of this work would have been possible. Client Insourcing was started from his invaluable insight, I was so excited that this work was gradually improved, and being acknowledged by research communities. I am very fortunate to be advised by him and I'm deeply indebted to him for enriching Ph.D. experience.

I would also like to thank my members of committee, Godmar Back, Xun Jian, Francisco Servant, and Walter Binder, for carefully reading my dissertation and presenting constructive feedbacks and valuable insight that made it possible for me to improve the overall quality of the dissertation research.

I also had great pleasure of working with members Software Innovations Lab. By working with Yin in a project, I could learn many techniques. Thanks to Breno, Karn, Jason for there helpful advice and kindness.

I would like to thank my family, especially - my parents, parents-in-law, sisters-in-law, brothers in-law, for their love, support, and encouragement. Last but not least, I dedicate this dissertation to my wife, Mihee Lee, and our lovely child, Aiden Sooha An, for their tireless love, support, and sacrifices.

[1]This material is based upon work supported by the National Science Foundation (NSF) under Grants No. 1650540 and 1717065. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF.

Contents

List of Figures

List of Tables

List of Abbreviations

CanDoR Catch and Release

JS-RCI JavaScript Refactoring Client Insourcing

Chapter 1

Introduction

The majority of modern computing applications are distributed in some way. A distributed computing application executes on different machines, connected to each other via a network. A distributed application's execution flows across the separate address spaces of its constituent parts that execute on different machines. One of the most common distributed architectures is *client-server*, in which the client remotely invokes services at the server, which executes them returning the results to the client. Distribution assigns which application component should run on which machine. Some distribution strategies are predefined; for example, user interfaces must display on the client, while a shared database must be placed on the server. Other distribution strategies aim at improving performance; for example, a powerful cloud-based server can execute some functionality faster than can a mobile device. Various middleware libraries and frameworks streamline the implementation of the remote interactions across the different parts of a distributed application.

As is the case for all software systems, developers commonly need to re-engineer distributed applications after deployment and usage. Re-engineering broadly captures evolutionary modifications that range from routine maintenance tasks to architecture-level changes [4, 18]. A re-engineering task can be as straightforward as fixing a bug, or can involve complex modifications that add a major feature, protect against security vulnerabilities, or remove performance bottlenecks. When performing re-engineering tasks, developers make use of various software tools—profilers, debuggers, refactoring browsers—commonly included with

modern Integrated Development Environments (IDE).

There is a major disconnect between modern software applications and mature software re-engineering tools. While most of modern applications are distributed, most of the widely available re-engineering tools are designed to work with centralized software, written to run on a single machine. Although some of these tools can be applied to distributed applications by treating their constituent parts as separate centralized applications, many of them fall short of fully addressing the whole range of re-engineering tasks that may need to be performed on modern distributed applications. For example, running a debugger only on the client part of a web applications may be insufficient to uncover certain performance bugs. Consequently, there is great potential benefit in creating software re-engineering tools that can support the unique characteristics of distributed applications.

This dissertation improves the state of the art in re-engineering distributed applications by introducing a novel software refactoring—*Client Insourcing*. We draw inspiration from business process re-engineering that can bring remote operations in-house via *Insourcing*. Once the insourced operations are redesigned and restructured, some of them can be outsourced anew. We apply to distributed applications the notion that local operations are easier to analyze and restructure than remote ones. Specifically, Client Insourcing automatically transforms a distributed application, comprising a client communicating with a remote server, to run as a centralized program. The resulting centralized variant retains to a large degree the semantics of the original application, but replaces all remote operations with local ones. The centralized variant becomes easier to analyze and modify not only because it has no remote operations, but as argued above, because the majority of program analysis and transformation approaches and tools have been developed for centralized programs. The centralized variant can be analyzed, modified, and verified to perform various re-engineering tasks. It can then be redistributed anew into a re-engineered distributed web application.

Client Insourcing is not a panacea that can resolve all difficulties of re-engineering modern distributed applications, which come in great variety. Nevertheless, we contribute to the state of the art by demonstrating that our approach can be successfully applied to re-engineering tasks that detect and fix performance bottlenecks, reshape the distribution granularity for dissimilar network conditions, and improve the responsiveness of web-based mobile apps. Our reference implementation targets so-called *full-stack JavaScript applications*, distributed applications in which both the client and server parts are written entirely in JavaScript. We take advantage of the monolingual nature of such applications to streamline our implementation and evaluation. Nevertheless, the applicability of our approach in general and of Client Insourcing in particular should not be limited to monolingual distributed applications. With additional engineering, our approach can be extended to work with multilingual applications by integrating language translation or virtual execution.

Summary of Contributions

In summary, this dissertation makes the following major contributions:

1. **The Client Insourcing Refactoring** The focal point of this dissertation is *Client Insourcing*, a new automatic refactoring that undoes distribution by gluing the local and remote parts of a distributed application together. Client Insourcing creates a centralized variant of a distributed application, replacing all remote invocations via middleware with local function calls. This local variant can then can be analyzed and modified more easily than its distributed counterpart. Then, the centralized variant can be redistributed with all the local modifications in place.

   We provide a reference implementation of Client Insourcing in the important domain of Full-Stack JavaScript applications. To demonstrate the applicability of Client In-

sourcing to facilitate common software maintenance and evolution tasks, we apply our reference implementation to perform adaptive and perfective tasks to applications in this target domain. The Client Insourcing and its reference implementation are presented in Chapter 3.

2. **A Debugging Approach for Web Applications** We applied Client Insourcing to introduce a debugging approach that can catch an important class of performance and resource consumption bugs in web applications. Localizing bugs in web applications is complicated by the potential presence of server/middleware misconfigurations and intermittent network connectivity. Our approach first applies Client Insourcing to convert a debugged application to its semantically equivalent centralized version, thus separating the programmer-written code from configuration/environmental issues as suspected bug causes. The centralized version is then debugged to fix various bugs. Finally, based on the bug fixing changes of the centralized version, a patch is automatically generated to fix the original application source files. The Debugging approach and its reference implementation are presented in Chapter 4.

3. **A Re-Architecting Approach for Correcting Ill-Conceived Distribution Granularity** We applied Client Insourcing to re-architect distributed applications, whose distribution granularity has turned ill-conceived. Distributed applications enhance their execution by using remote resources. However, distributed execution incurs communication overheads; if these overheads are not offset by the yet larger execution enhancement, distribution becomes counterproductive. For maximum benefits, the distribution's granularity cannot be too fine or too crude; it must be just right. To adjust the distribution of such applications, our approach automatically reshapes their remote invocations to reduce aggregate latency and resource consumption by means of a series of domain-specific automatic refactorings. The re-architecting approach and

its reference implementation are presented in Chapter 5.

4. **An Approach to Improving the Responsiveness of Web-based apps** We applied Client Insourcing to create a dynamic approach to redistributing full-stack JavaScript mobile apps to improve their responsiveness. A mobile web app's performance depends heavily on the underlying network and the client device's hardware capacity. Static distribution allocates some application functionality to run on the client and some on the server, with the resulting allocation remaining fixed throughout the app's execution. If the available network and the client device mismatch those assumed during the static distribution, app responsiveness and energy efficiency can suffer. By monitoring the network conditions, our framework determines whether a dynamic redistribution would improve application performance. It then triggers a redistribution that moves server-side functionalities to the mobile client and vice versa. When our framework moves server-side functions and their reachable state to the client, they are invoked as regular local functions. The moved functionalities are accessed remotely again as soon as the network conditions improve. The re-architecting approach and its reference implementation are presented in Chapter 6.

Structure

The rest of this Ph.D. dissertation is organized as follows. Chapter 2 presents the technical background of this dissertation research. Chapters 3, 4, 5, and 6 cover our approach and reference implementations. The chapter 8 discusses yet another possible application of Client Insourcing: facilitating the integration of edge computing and storage resource into exiting client-cloud applications, transitioning full-stack JS apps. Chapter 9 presents concluding remarks and summarizes the contributions of this dissertation.

Chapter 2

Background

This sections introduces the main concepts and technologies used by this dissertation: software maintenance, refactoring, and, distributed Web applications. The reader familiar with these topics might choose to skip the corresponding sections.

2.1 Software Maintenance

Software Maintenance is an important part of the Software Engineering of life cycle, concerned with various modifications of a software product after its initial delivery. It has been determined software maintenance and evolution incur the majority of software engineering effort and costs [16, 19, 58]. The ISO/IEC 14764 standard divides software maintenance into four categories: *adaptive*, *corrective*, *perfective*, and *preventive*, as depicted in Figure 2.1. Each maintenance category involves modifying a software system, but for dissimilar purposes. *Corrective* modifications fix bugs and other defects. *Adaptive* modifications change a software system, so it could be operated in a new environment, such as different architectures or operating systems. *Perfective* modifications enhance a software system with new functionalities. *Preventive* modifications are intended to avoid the introduction of defects and other imperfections in the future.

Modifying distributed applications to perform any kind of maintenance is particularly hard and error-prone. Distributed applications have complex control flows, affected by the con-

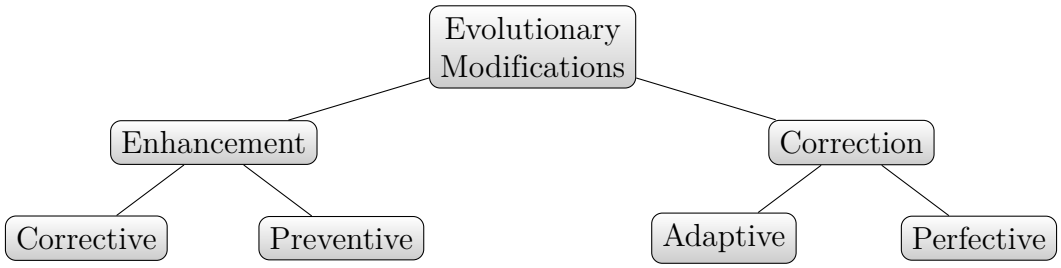


Figure 2.1: Category of Evolutionary Software Modifications

ventions of the used middleware libraries and frameworks. The execution and performance of distributed applications can be affected by server/middleware misconfigurations and network volatility. These mainstay conditions of distributed execution hinder the most important phase of the maintenance process: determining the exact causes of problems to solve and the locations in the code that implement them.

2.2 Refactoring

Refactoring is a program transformation that changes the original source code while preserving its semantics. Refactoring has become an important part of software maintenance, applied widely to improve modularity, complexity, maintainability, and efficiency. Some refactoring techniques have become common, and their automatic application is supported by modern integrated development environments (IDE). For instance, *Rename Function* and *Rename Variable* are the simplest refactoring that can be applied to avoid naming conflicts and create more meaningful names for program functions and variables, respectively. The *Extract Function* refactoring extracts a block of code into a separate function declaration. A preventative maintenance task could apply the Extract Function refactoring to replace duplicate parts with calls to the function that contains the extracted duplicated code. Another preventative maintenance task can be concerned with eliminating very short functions that

are unnecessary and hinder comprehensibility. The *Inline Function* refactoring can be seen as the reverse version of *Extract Function*: it replaces a function call with the code of the called function (Figure 2.2).

The vast majority of existing refactoring transformation and their automated support work only with centralized applications. Existing refactoring transformations can be applied to individual parts of a distributed application, but not to the portions that contain distributed control flows. Given that the majority of software is distributed in some fashion nowadays, there is great potential benefit in extending the benefits of refactoring to distributed software.

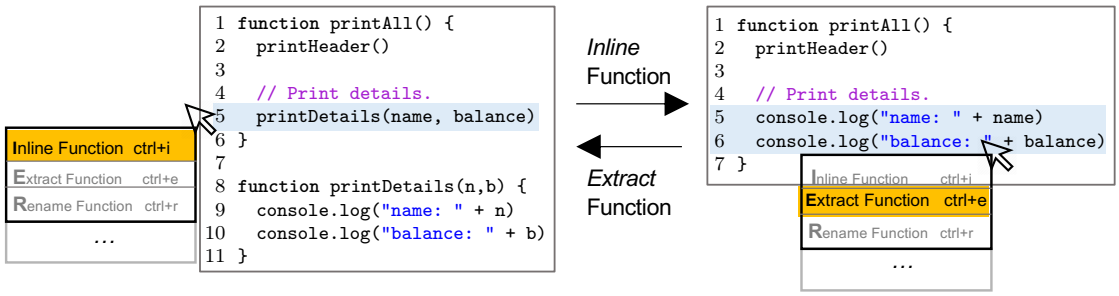


Figure 2.2: *Inline Function* and *Extract Function* Refactorings

2.3 Distribution and Distributed Execution

By using remote resources, a distributed application enhances its functionality or improves its quality of service. Sometimes distribution is inevitable, when certain resources can be accessed only remotely. In other cases, distribution is a choice: the same functionality can be executed by means of local or remote resources. Executing a functionality remotely reduces the computational load on the client at the cost of the delay, measured as the remote invocation's latency, and the local resources consumed to make this invocation. Typically it is distribution middleware that consumes these additional resources, including computation, memory, and energy. The invocation latency measures the expected deterioration in the user

experience—the more time the user has to wait for a remote functionality to complete, the less satisfying their experience will be. Hence, the distribution cost is the sum of the expected deterioration in the user experience and the amount of additional local resources consumed to invoke the remote functionality. These costs must be offset by the attendant execution enhancement for distribution to remain beneficial. Otherwise, distributed execution only incurs overheads, which hurt both performance and maintainability. Of course, if some functionality can be accessed only remotely, these overheads are justified and unavoidable. However, if distribution is introduced to improve an application's quality of service, it must be introduced at the right level of granularity, when opting to execute a functionality remotely over the network indeed improves application performance (Figure 2.3). Introducing *too much distribution* for the expected benefits is a known architectural problem, documented as the *Nano-Service Anti-Pattern* [76].

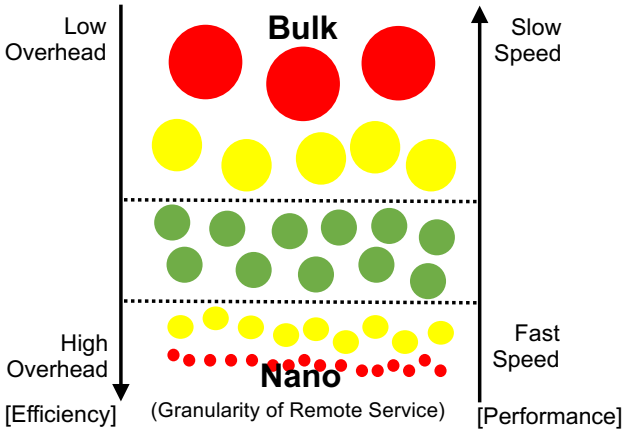


Figure 2.3: Level of Distribution: Performance versus Efficiency

Distributed execution is often used to improve performance. For example, in mobile apps, as the computing resources of remote, cloud-based servers surpass those of mobile devices, a functionality can be executed faster remotely than locally. However, executing a remote cloud-based functionality requires passing parameters and receiving results over the network. Network communication significantly complicates the device/cloud performance equation.

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, as operating over high-loss networks requires re-transmission, which consumes additional battery power. Hence, the added overhead of network transfer burdens the mobile device's battery budgets, often negating the performance benefits of executing a local functionality at a remote cloud-based server [99].

Distributed web applications, structured around the ubiquitous REST (Resentational State Transfer) architecture, invoke web services by means of the HTTP (Hypertext Transfer Protocol) protocol. Remote web services are referenced by their URI (Uniform Resource Identifier); they then respond by sending back typically text-based documents, formatted either in HTML (Hypertext Markup Language), JSON (JavaScript Object Notation), or XML (extensible modeling language). Rest APIs are typically stateless, meaning that each request can be handled in an isolated manner. That is, the server does not store any state information about the client session. However, stateful session-specific information can be maintained by means of persistent storage, realized as files or databases.

The aforementioned distributed Web architecture makes it possible for clients and servers, implemented in different programming languages under dissimilar platforms, to seamlessly interact with each other. The communication between clients and servers is typically facilitated by various middleware systems, exposed to programmers as libraries and frameworks. Remote functionalities are invoked by means of distribution middleware, which exposes APIs whose protocols and conventions differ from invocations in the same address space. These middleware APIs tend to differ widely depending on their vendor.

One of the key functions of a middleware system is marshaling and unmarshaling method parameters and return values. Marshaling converts program values to a data format suitable for transmission; unmarshaling reverses the process by converting the transmitted data

format to regular program values (See Figure 2.4).

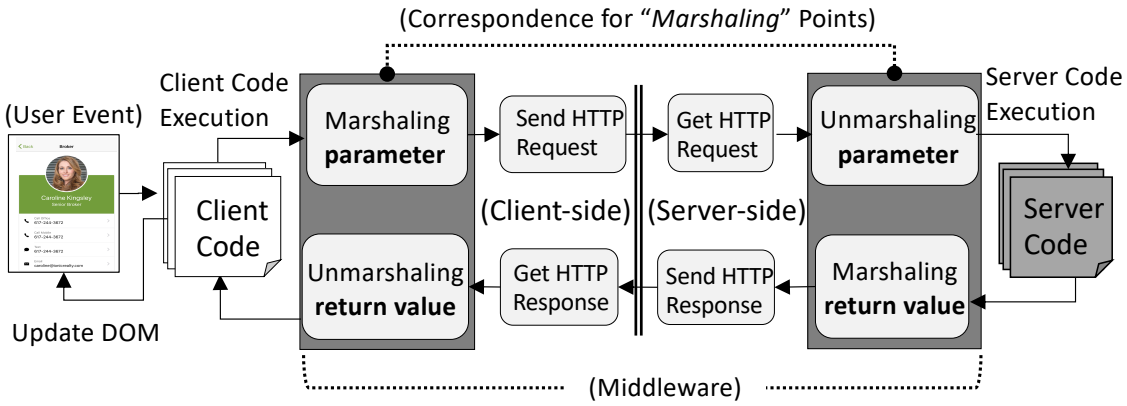


Figure 2.4: Web App's Execution Model

Chapter 3

Client Insourcing Refactoring

Developers often need to re-engineer web applications to address requirement changes made only after deployment and usage. Re-engineering captures evolutionary modifications that range from maintenance tasks to architecture-level changes [18]. A re-engineering effort can involve adding a major feature, protecting against security vulnerabilities, or removing performance bottlenecks. Modifying existing web applications requires complex program analysis and modification operations that are hard to perform and even harder to verify. One of the main causes of this complexity is the distributed execution model of web applications.

In this model, a web application's execution flows across the separate address spaces of its client and server parts. All remote interactions are typically implemented by means of middleware libraries. As a result, the control flow of web applications can be highly complex, with their business and communication logic intermingled. That complexity hinders all tracing and debugging tasks. In addition, distributed execution over the network makes web applications vulnerable to partial failure and non-determinism.

Program analysis is central to software comprehension. The web is predominated by dynamic languages, which defeat static analysis techniques. Hence, to comprehend programs written in dynamic languages, such as JavaScript, requires dynamic analysis. Software debugging hinges on the ability to repeat executions deterministically [63, 73]. However, many web applications are stateful, with certain client server interactions changing the server's state. It can be quite laborious and error-prone to restore the original state to be able to repeat

a remote buggy operation [39, 59, 77]. All in all, it is the presence of both distribution and stateful execution that makes it so hard to trace and modify web applications.

In this paper, we draw inspiration from business process re-engineering that can bring remote operations in-house via *insourcing*. Once the insourced operations are redesigned and restructured, some of them can be outsourced anew. As argued above, the notion of local operations being easier to analyze and restructure than remote ones equally applies to web applications.

Specifically, the approach presented herein first automatically transforms a web application, comprising a client communicating with a remote server, to run as a centralized program. The resulting centralized variant retains to a large degree the semantics of the original application, but replaces all remote operations with local ones. The centralized variant becomes easier to analyze and modify not only because it has no remote operations, but also because the majority of program analysis and transformation approaches and tools have been developed for centralized programs. After the centralized variant is modified to address the new requirements and the modifications have been verified, it is then redistributed again into a re-engineered distributed web application. Our target domain are web applications written entirely in JavaScript, both the client and server parts; such applications are referred to as *full-stack JavaScript applications*. We take advantage of the monolingual nature of such applications to streamline our implementation.

In a web application, clients communicate with the server by means of the HTTP protocol, typically in a request/response pattern. However, from the implementation perspective, the HTTP functionality can be supported by a variety of middleware libraries with vastly dissimilar APIs [67]. To be able to identify and replace the HTTP communication functionality, a web application may need to be executed multiple times under different inputs. However, some remote interactions cause the server to change its state. For example, a client can pass

a parameter to the server, which would store that parameter in the server-side database. In addition, the non-database state can change as well (e.g., adding the parameter to the JavaScript list of displayed items). In different states, the server may respond dissimilarly, thus making it impossible to identify HTTP middleware API calls, so they can be correctly replaced with corresponding local calls of the insourced functionalities.

The focal point of our approach is Client Insourcing, a new automatic refactoring that undoes distribution by gluing the local and remote parts of a distributed application together. Our approach can precisely identify the functionality of HTTP middleware—irrespective of its API and in the presence of stateful operations—by combining program instrumentation, profiling, and fuzzing in a novel way. Our ideas are realized in our reference implementation—Java Script Remote Client Insourcing (**JS-RCI**). We evaluate our approach's value, correctness, and utility by applying **JS-RCI** to re-engineer a set of real-world web applications.

The rest of this chapter is structured as follows. Section 3.1 motivates and summarizes our approach. Sections 3.2 and 3.3 present the design and implementation specifics of the Client Insourcing refactoring, respectively. Section 3.4 reports on how we applied Client Insourcing to streamline three representative re-engineering scenarios of web applications. Section 3.5 discusses various applicability issues pertaining to our approach.

3.1 Re-engineering Distributed Web Apps

Developers often find themselves having to re-engineer an actively used application to ensure its continued utility, reliability, and safety. When interacting with an application in real-world settings, users may discover and report inefficiencies and imperfections. Users may request that new features be added to an application to increase its utility. As users discover existing faults and request new features, developers can decide to re-engineer the application

to deliver an improved version to the users. Re-engineering modifications can range from routine maintenance and evolution tasks to major architectural transformations. Next, we demonstrate two examples of re-engineering full-stack JavaScript applications.

3.1.1 Example Apps

The following code snippets come from two third-party full-stack JavaScript applications *realty-rest* (Figure 3.1) and *recipebook* (Figure 3.2), with both of their client and server parts shown. Both applications rely on the network for their client and server parts to communicate with each other. The primary user base of *realty-rest* are real-estate brokers, licensed professionals that sell and purchase various properties on behalf of their clients. Due to the nature of their business operations, real-estate brokers lead highly mobile professional lives, moving from location to location to show properties to potential buyers. Hence, as a mobile app, *realty-rest* is well-aligned with the needs of its users, who rely on the app to be readily available, responsive, and reliable. To start using the app, a user selects a property from the list of all properties registered with the system. The selected property can then be updated or deleted, with the app's client then sending HTTP commands to the server, (e.g., DELETE /property/favorite to remove a property from the list of favorites, etc). The HTTP commands are wrapped into distribution middleware (angular2/http) with JavaScript API. Specifically, the client invokes `HTTP.delete` passing a URL parameter, with angular2/http delivering the invocation to the server and calling function `unfavorite` there. This function finds and deletes the passed `property`, returning the updated list of favorites to the client. angular2/http marshals both `property` and the result-to-return as JSON-encoded messages. The client unmarshals the returned result to update the GUI. The *recipebook* maintains a list of cooking recipes at the server, so different clients could retrieve and update the maintained recipes. *recipebook* uses a different middleware library to wrap its HTTP

commands—angularJS, whose JavaScript API differs from that of angular2/http. While *realty-rest* is a two-tier app (JavaScript client and sever), *recipebook* is three-tier (adding a database tier).

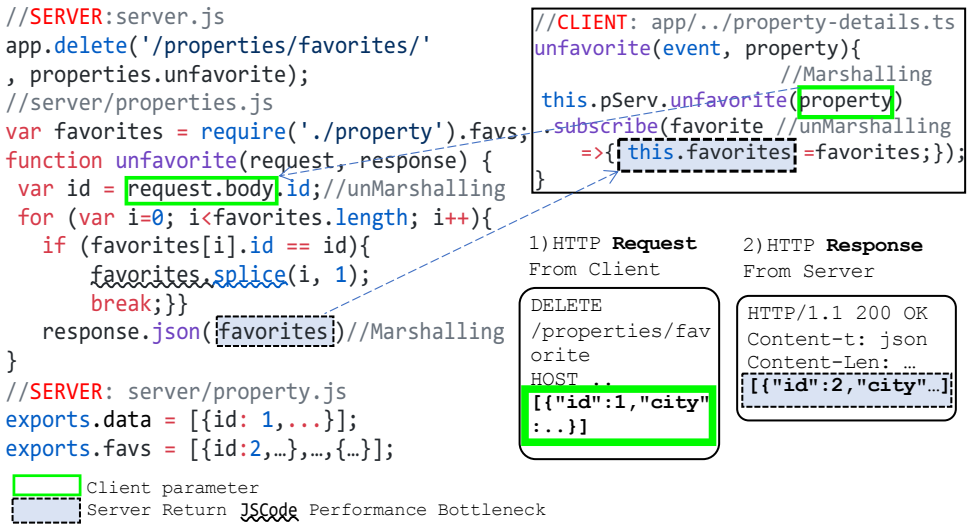


Figure 3.1: the *realty-rest* app's highlighted client and server code

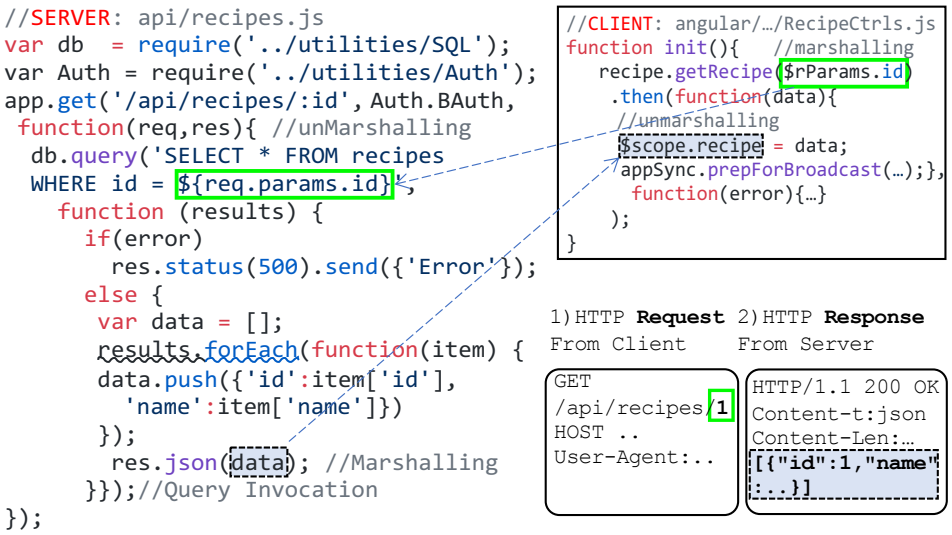


Figure 3.2: The *recipebook* app's highlighted client and server code

Next, we present examples of how *realty-rest* and *recipebook* may need to be re-engineered to address new requirements.

3.1.2 Adapting to Disconnected Operation

Examining the history of *realty-rest* reveals that some of this app's functionalities have been moved between its client and server sites[1]. Since scant documentation makes it hard to ascertain the reason for these moves, we next discuss a typical new feature that enables distributed apps to continue operation in the absence of a network connection. In particular, if users need to operate a mobile app in locations with limited or intermittent network connectivity, the app has to deliver its core business functionality without relying on any remote services. To enable such offline operations, several strategies have been proposed [74]. One such strategy is *replication*, which replicates a remote component locally, so the local copy acts as a proxy of its remote counterpart. A consistency protocol keeps both copies in sync. A naïve strategy for replicating a remote functionality would be just to copy its complete source files to the client, adapting the copied code by hand as necessary. However, such complete copying unnecessary replicates functionalities, some of which become "dead code."

3.1.3 Enhancing Privacy

Enterprises often find themselves in need to enhance user privacy in a released application. Consider a request to keep the *realty-rest* user's property browsing histories private from other real-estate brokers due to business competition reasons. To ensure user privacy, certain server-side functionalities (e.g., Customer Relationship Management (CRM)) can be redistributed to a special server that requires authentication before giving access to sensitive information. In fact, *realty-rest* indeed has gone through a similar modification, as evidenced by the existence of *realty-salesforce*[2], which provides the same business function-

[1] ionic2-realty (https://github.com/ccoenraets/ionic2-realty)
[2] realty-salesforce (https://github.com/ccoenraets/ionic2-realty-salesforce)

ality, but takes advantage of third-party trusted identification and security features. To re-engineer *realty-rest* into *realty-salesforce*, programmers would have to identify and migrate the relevant functionality to another server, modifying the client to communicate with different servers (regular and secure).

3.1.4 Improving Performance

If a substantial subset of users becomes unsatisfied with application performance, programmers may be asked to identify and remove performance bottlenecks. The left side of Figure 3.1 displays the server function [language=JavaScript,frame=none,numbers=none]unfavorite, which contains a known performance bottleneck, rooted in the usage of `favorite.splice(i,1)`, an inefficient API for removing collection items. In fact, an actual pull request[3] states that `Array.splice()`'s performance is between 1.5 and 10 times slower than that of a customized implementation, comprising a `for` loop iteration and `Array.pop()`. To be able to identify this particular source of the experienced performance bottleneck, programmers either would have to be intimately familiar with the peculiarities of JavaScript APIs or to rely on detailed execution profiling, typically available only for centralized programs. *recipebook* also contains a similarly inefficient `forEach` loop[4]. Notice that the distributed control flow that invokes these inefficient functions, starting from the graphical actions at the client, traversing the network through layers and layers of middleware, and finally executing the functions at the server. The invocation flows can be interrupted by network volatility and authentication failures. Hence, it is both complex control flows and possible failures that make it hard to isolate the performance of a web application's function.

[3] Perfective Modification for Array.splice() (https://github.com/nodejs/node/pull/20453)

[4] A modification request to remove this inefficiency appears here: https://github.com/elastic/apm-agent-nodejs/pull/1275

3.1.5 Client Insourcing to the Rescue

Next, we explain how Client Insourcing can facilitate the re-engi neering tasks outlined above.

Redistribution Client Insourcing creates a redistributable centralized variant devoid of the unnecessary middleware functionality. Once the variant is modified, it can be redistributed automatically. Numerous complementary research efforts have focused on automating the process of distributing centralized applications, with automatic transformation tools released to the public [44, 52]. Because the majority of existing refactoring techniques are designed for regular centralized applications, they can be applied at will to centralized variants. For example, the *Extract Function* refactoring can be used to separate some privacy-sensitive code within a function into a separate function to be executed in a different environment. After the sensitive code portions are separated into their own encapsulation units, the resulting program can be redistributed, placing the sensitive units to execute in separate privacy-enforcing server environments.

Isolated Profiling What if business logic can be precisely isolated from middleware and distribution-related functionality? Then the isolated code can be easily profiled to ascertain its performance characteristics and identify any performance bottlenecks. Client Insourcing enables such isolated profiling by removing middleware and gluing the remote parts of a web application together.

Offline Operation Client Insourcing can enable offline operation, without copying any unnecessary code from the server to the client, by replicating only the remote functionality's subset needed at the client. The replicated subset can include both JavaScript code and data persisted in a database.

3.2 Design & Reference Implementation

In this section, we explain our design options and then detail the specifics of our implementation of the Client Insourcing refactoring.

3.2.1 Design Overview

We give an overview of the main design decisions behind Client Insourcing via specific examples. Consider the task of moving the server functionalities of DEL /favorite or GET /recipe/
:id to execute at the clients (Figure 3.1). Instead of invoking these functions via middleware that handles communication, partial failures, and authentication, they would become regular local functions to be called directly. Hence, all middleware-based code would have to be replaced with direct function calls.

Consider the service DEL /favorite, whose business logic is encapsulated within the server-side `unfavorite` function. We want to insource `unfavorite` so it can be called as a regular local function. However, we cannot simply move this function from the server to the client, as its business logic and middleware functionality are intermingled. In addition, the `exports.favorite` array, referenced in the body of `unfavorite`, is declared externally. If `unfavorite` and `exports.favorite` are not moved together, invoking the function locally would raise an error. Hence, we must move all the referenced externally declared program elements to the client as well. **JS-RCI** identifies the exact boundaries of the server functionality to insource. However, some dependent business logic of GET /recipe/:id is not confined to JavaScript code only. **JS-RCI** also transparently insources code that persists data in a relational database.

```
//app/../property-details.js
import {j5ga2} from './j5ga2';
unfavorite { const IS_SYNC = false;
  if (IS_SYNC) {//synchronous call
    this.favorites = j5ga2(property.id);
    return;
  } //default: non-blocking call
  new Promise((resolve,reject) => {
    var out_j5ga2 = j5ga2(property.id);
    resolve(out_j5ga2);
  }).then(res => this.favorites = res);
}
```

```
//app/../b8f9a.js
exports.favs = [{id: 1,city:'B,..}];
//app/../j5ga2.js
var favorites=require('./b8f9a').favs;
export function j5ga2(input){
 var tmpv1 = input;
 var id = tmpv1;
 for (var i=0; i< favorites.length;
i++){… favorites.splice(i, 1);…}
 tmpv0 = favorites;
 var output = tmpv0;
 return output;}//extracted function
```

Figure 3.3: Insourced a functionality: DEL /properties/favorite in *realty-rest*

3.2.2 Identifying the Code to Insource

Next, we present our solution for automating the steps above, realized as the *Client Insourcing Refactoring*. One of our design goals was to make sure that this domain-specific refactoring is not too burdensome for the programmer. We assume that the refactored applications come with a set of standard test cases, and that the application of these cases is automated. It is during the application of such test cases, when **JS-RCI** detects the marshalling/unmarshalling points of the functionality to insource at the client invocations. Intuitively, the purpose of detecting these marshalling/unmarshalling points in the client code is to identify the entry/exit execution points of the remote functionality to insource. These points correspond to the locations in the client code, at which remote invocation parameters are marshalled to be transferred across the network, and the remote invocation's results are unmarshalled to be used in the subsequent client execution.

To extract all the server code of the remote functionality to insource, **JS-RCI** uses symbolic execution. We assume that the server is implemented in Node.js and define the execution rules as pertaining to this framework's architectural conventions. First of all, **JS-RCI** normalizes server code to facilitate to detect entry/exit execution points and extract the executed JavaScript code. To that end, **JS-RCI** additionally introduce temporal local variables and makes JavaScript Statement to have a single operation (i.e., `tmpv0` and `tmpv1` in Figure 3.3).

For symbolic execution, we use z3 [24], parameterized with our own set of rules and facts. For example, the profiled parameters and return results of a remote functionality are added as new z3 facts. Figure 3.5 shows the overall process of Client Insourcing.

3.2.3 Exploiting Asynchrony

Notice that in a distributed client-server application, the remotely invoked functionalities running at the server, and the client code invoking these functionalities, run in separate address spaces that are not shared (unless the application runs on top of some distributed shared memory system [84], which is not a standard option for web applications). The parameters passed to remote invocations and the invocation results are copied between the client and the server heaps, always creating a new copy rather than mutating any existing program state. Hence, in a distributed application that uses application-layer middleware (e.g., HTTPClient), the client and the server parts share *no mutable state* (See Figure 3.4). Following this observation, one can conclude that the client and the server parts have *no non-middleware dependencies* between them. That is, in such distributed applications, the only way for the client code to invoke a server-side functionality is by making a remote invocation via middleware. To maintain this semantics, our design also provides a single entry point to invoke the insourced functionality, a function previously invoked via a middleware API call at the server. It is these insights that make it possible to safely execute the insourced code asynchronously, without any need for synchronization! Our design of Client Insourcing takes advantage of these insights by executing the insourced functionality *asynchronously by default*. In particular, the generated code makes use of the `Promise` framework that exposes asynchronous execution via a standardized interface that uses the programming idioms congruent with the design of JavaScript.

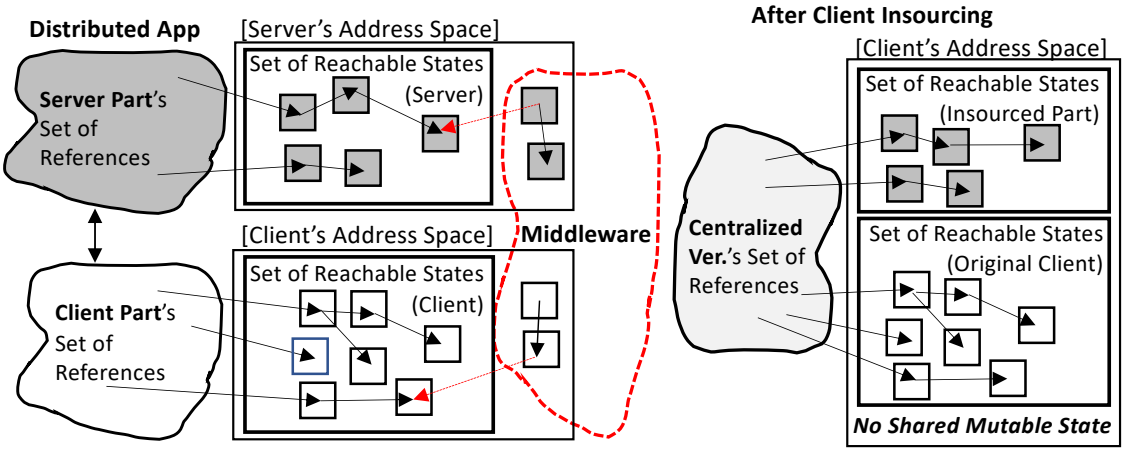


Figure 3.4: Reachable States between Server and Client parts

For a specific example, consider the code listing in Figure 3.3 that shows the generated client code for `DEL /favorite`. Notice that the default invocation model for this insourced function is asynchronous, a runtime behavior that is put into effect by creating a new instance of a `Promise` closure. Once the asynchronous execution of `j5ga2` completes, the *Promise* framework invokes the callback `resolve` to handle the successful execution. Since our design aims for versatility, we provide an option for the insourced functionality to be invoked synchronously as a regular blocking local call. This behavior can be put into effect by setting the value of the `boolean` variable `IS_SYNC` to `true`.

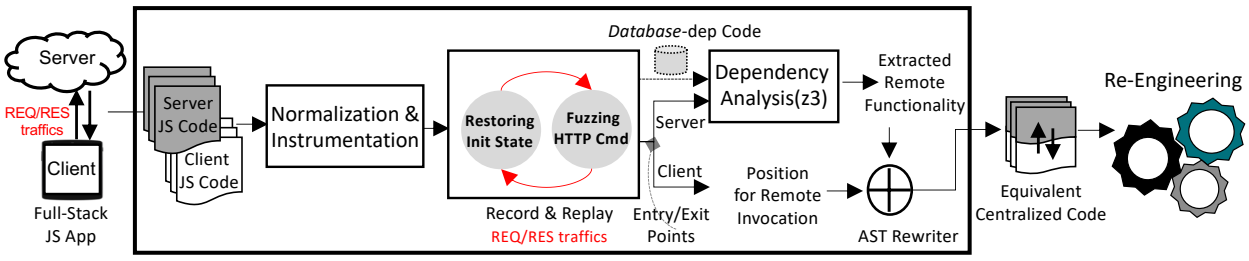


Figure 3.5: Overall process for Client Insourcing

3.3 Implementation Specifics

In this section, we provide some additional details pertaining to our implementation choices.

3.3.1 Detecting Marshalling Points in Client/Server Program

In a full-stack JavaScript application, the client interacts with the server in the request/response pattern, exchanging data in JSON or XML formats. Client Insourcing determines which middleware API calls send and receive the HTTP protocol commands through the following automatic and application-agnostic procedure.

First, the round-trip traffic of the client/server interactions is recorded. Then, **JS-RCI** parses the request/response data to obtain the deserialized values of *client parameters* and *server return*. To that end, **JS-RCI** captures live network traffic, not only to record/replay the HTTP interactions, but also to extract the used HTTP commands. To capture business logic (as compared to fault handling logic), **JS-RCI** only processes the responses with the status code of 400 (i.e., successful execution).

Next, **JS-RCI** replays the recorded round-trip execution that invokes the remote functionality to insource. Both the client and server parts are dynamically instrumented to keep track of values for (1) arguments and returns of the function invocations (2) reading and writing variables. **JS-RCI** keeps comparing the values of the invocations and variables to identify the ones equal to *client parameter* and *server return*. To instrument the invocations and variable accesses, **JS-RCI** uses the Jalangi2 callback APIs [88].

To identify the entry points at the server, **JS-RCI** keeps comparing the values for recorded *client parameter* of the remote functionality. That is, the parameter has been unmarshalled and is about to be used. To identify the exit point at the server, **JS-RCI** follows a similar

procedure, but looks for the value recorded as the *server return* of executing the remote functionality. Finding an equal value read or written determines the exit point of the remote functionality. That is, the return value is about to be marshalled and sent across the network to the client. One may wonder: how does our approach determine that the equality comparison indeed identifies the entry and exit points of the remote functionality rather than some intermediate values that also happen to be equal to the values of *client parameter* and *server return*? To identify the entry and exit points at the server, our analysis identifies the first instance of the *client parameter* equality and the last instance of the *server return* value equality. Unlike its server-side logic, the analysis identifies the last instance of the *client parameter* equality and the first instance of the *server return* value equality.

Fuzzing Request/Response Messages

Even with these arrangements in place, it is still possible to misidentify the correct entry and exit points, particularly if the parameters or return results are primitive types, such as built-in numbers or strings (i.e., 0 or 1 values of id in `findById` service). To prevent such misidentification, **JS-RCI** populates the original round-trip content by padding the HTTP header and body data with random bits. A fuzzing dictionary is also applied to fuzzable primitives types: string has the possible values "JSRCIStr" and integer has the possible values from "90,000" and to "100,000." For instance, **JS-RCI** encodes "1" as "90,001". For a service without a client parameter (i.e., `findAll` type services), **JS-RCI** fuzzes the request with "JSRCIStr" so **JS-RCI** can locate the function block's begin as the entry point.

Achieving the Idempotency for Record/Replay Executions

Despite the stateless nature of the RESTful architecture that guides the design of WWW, few realistic web applications are truly stateless. In fact, every HTTP request can change the server's state. These changes hinder the precision of our detection of the server's marshalling points, introducing *false-negatives*. Even HTTP traffic were replied with identical requests, a stateful server is likely to behave differently in 1) marshalling its response output or 2) entering the remote functionality through a different point (e.g., if a visited entry is deleted, it cannot be revisited).

Testing web applications deterministically requires that test cases be isolated [39, 77]. Otherwise, the same test case can yield dissimilar results when executed with the same input. Restoring the server to its original state by hand would be expensive in realistic web applications, requiring a manual reset of the relevant database tables and a fresh restart of the server. In contrast, **JS-RCI** fully automates the process to achieve the idempotent execution of all HTTP requests. To maintain the original server's state, **JS-RCI** interleaves an automatically generated *restore* operation, run between all successive record or replay executions. Similarly to a prior approach that checkpoints PHP web application [39], **JS-RCI** initiates the restore operation with a special HTTP request. Similarly to manipulating fuzzed request messages, **JS-RCI** generates the *restore* operations by enhancing original HTTP requests with the new "JSRCIRestore" parameter. To be able to restore the server state, **JS-RCI** first saves the initial values of all server's global variables, so they can be restored on demand. Also, as part of its restore operation, **JS-RCI** executes transaction control operations between every SQL invocations, so the database rollbacks to its previous state.

As its specific implementation strategy, **JS-RCI** uses jalangi2, whose *shadow execution* instruments the original JavaScript code, so the server events can be hooked dynamically.

First, **JS-RCI** detects all (1) post declarations of global variables (g) and (2) pre/post *Call Expressions* of SQL statements (f). Then, it uses two customized shadow executions at (1), $g' = store(g)$ to serialize and store the state of all global variables and $restore(g, g')$ to reset all global variables to their original values, hooked by *restore* HTTP commands. To restore the database state, **JS-RCI** uses shadow execution $invoke(f, sql\_stat)$, which invokes *Call Expression* of a SQL statement f with a new SQL clause as the argument. $invoke(f,$"Start TRANSACTION") and $invoke(f,$"ROLLBACK") are executed at pre and post invocations of f, respectively. **JS-RCI** executes these operations only once for the nested SQL invocations.

3.3.2 Identifying the Relevant Server Code

One of the factors that complicates the *Client Insourcing* refactoring is that the code comprising the functionality of the insourced functionality may not be confined to the boundaries of a single function or even the same script. While the entry point of the remote execution can be a JavaScript function, this function can be invoking other functions or reference variables declared elsewhere. When insourcing a remote functionality, all this dependent code must be moved together to the client to create a self-sufficient local call that no longer relies on any server-based code.

To determine the data dependencies between the entry/exit points of a distributed application's remote functionality, we draw lessons provided by the state-of-the-art JavaScript analysis frameworks [40, 41, 98]. JSdep [98] logically hypotheses a Data-Dep relation between JavaScript statements based on read/write facts, a point-to-analysis model of GateKeeper [40] and a control flow analysis [41] . For instance, an assignment statement Assign becomes a fact that implies Read and Write relations for the variables involved. Read and Write on the same variable between different statements imply a Data-Dep relation at the

statement level.

$$
\begin{aligned}
\textsc{Assign}(stmt_1,v_1,v_2) &\quad \text{//var } v_1 = v_2;\ v \text{ is variable, } stmt_1 \text{ is statement}\\
\textsc{Write}(stmt_1,v_1) &\quad \leftarrow \textsc{Assign}(stmt_1\ v_1,v_2)\\
\textsc{Read}(stmt,v_2) &\quad \leftarrow \textsc{Assign}(stmt,v_1,v_2)\\
\textsc{Data-Dep}(stmt_1,stmt_2) &\quad \leftarrow \textsc{Read}(stmt_1,v_1) \wedge \textsc{Write}(stmt_2,v_1)\\
&\quad \ldots
\end{aligned}
$$

We extend JSdep's knowledge base with the rules and facts, necessary to model the execution of middleware-based statements. In particular, we define the UNMARSHAL/MARSHAL rules to identify the entry and exit points, whose WRITE clauses are inferred from the logged profiling data. To that end, JS-RCI encodes the REF facts by using the logged values to symbolically copy the unmarshalled/marshalled values ($V^{u_{id}}_{unMar}$/$V^{u_{id}}_{Mar}$, u_{id} is an unique execution id such as "J5ga2") into the local variables as follows:

$$
\begin{aligned}
&\text{//the entry point at the server}\\
&\textsc{UnMarshal}(stmt_1,v_{unMar},\ V^{u_{id}}_{unMar}) \quad \leftarrow \textsc{Write}(stmt_1,v_{unMar}) \wedge \textsc{Ref}(v_{unMar},V^{u_{id}}_{unMar})\\
&\text{//the exit point at the server}\\
&\textsc{Marshal}(stmt_1,v_{Mar},\ V^{u_{id}}_{Mar}) \quad \leftarrow \textsc{Write}(stmt_1,v_{Mar}) \wedge \textsc{Ref}(v_{Mar},V^{u_{id}}_{Mar})
\end{aligned}
$$

Based on the resulting knowledge base, JS-RCI can query the executed statements $stmt_n$ for the presence of unmarshalled/marshalled values. Predicate EXECUTEDSTMTS is a conjunction of two clauses: the first clause expresses the dependent statements for the parameter marshalling statement, while the second clause expresses the dependent statements for the result unmarshalling statement, both specific to the server execution. Because the DATA-DEP relation is transitive, one can obtain the executed statements from the entry/exit points, as expressed by the following set operations:

$$
\begin{aligned}
\textsc{ExecutedStmts}(stmt_n,V^{u_{id}}_{unMar},\ V^{u_{id}}_{Mar}) \quad &\leftarrow\\
(\textsc{Data-Dep}(stmt_n\ ,stmt_1) \wedge \textsc{Marshal}(stmt_1,v_1,\ V^{u_{id}}_{Mar})) \wedge&\\
(\neg\textsc{Data-Dep}(stmt_n,stmt_2) \wedge \textsc{UnMarshal}(stmt_2,v_2,\ V^{u_{id}}_{unMar}))&
\end{aligned}
$$

3.3.3 Insourcing Database-Dependent Code

Our approach can also insource code that persists data in a relational database. To that end, we take advantage of the ubiquity of SQL. Recall that **JS-RCI** dynamically instruments string values used as arguments and return values in all function calls. To identify the entry point for database-related operations, **JS-RCI** examines the function calls whose strings arguments represent the CRUD operations (Create, Read, Update, and Delete). Consider the code snippet in Figure 3.1. **JS-RCI** detects that the following *Call Expression* is a READ operation, as it is a SQL SELECT statement:

```
db.query("SELECT * FROM recipes WHERE id=id", function(result)..);
```

Although the server and the client are written in JavaScript and their respective database engines accept the same SQL statements, the JavaScript APIs of these engines differ. So it would be impossible to simply move this *Call Expression* and its dependent statements (e.g., `var db = require("../utilities/SQL");`) to the client. Hence, **JS-RCI** adapts the server-side database API to that of the client rather than copying the database-specific statements verbatim. With these API calls translated, developers can simply migrate the server-side data schema and tables. Notice that database engines store their data in dissimilar proprietary formats.

As a specific example, consider how **JS-RCI** translates the database API calls of MySQL[5] to those of *alasql*[6].

By extracting the arguments and return values of function calls, **JS-RCI** extracts table names and their columns, thereby inferring a complete data schema of the insourced code. Extracting the actual table content requires a different approach, as the WHERE clause and numer-

[5]https://github.com/mysqljs/mysql
[6]https://github.com/agershun/alasql

ical functions, such as COUNT, return only a subset of table rows. To retrieve all database data, **JS-RCI** instruments the server code by using the shadow execution *invoke*(`db.query`, `"SELECT * FROM recipes"`), which is introduced in Section 3.3.1. To infer the database schema from the extracted entries, **JS-RCI** uses tableschema-py[7]. Finally, **JS-RCI** uses the CREATE and INSERT commands with *alasql* to create tables and insert the extracted data into them, respectively, for the client-side database.

3.4 Evaluation

To determine how feasible and useful our approach is, we conduct an empirical evaluation driven by the following questions:

- **RQ1. Effort Saved by Client Insourcing :** How much programmer effort is saved by applying **JS-RCI**? We measure the saved effort as the number of lines of code that would need to be copied and modified by hand. **JS-RCI** saves this effort automating these manual source code changes. (Section 3.4.2)

- **RQ2. Correctness of Client Insourcing :** Does Client Insourcing preserve the business logic of full-stack JavaScript applications? Are existing standard use-cases still applicable to the centralized variants of the subject applications? (Section 3.4.3)

- **RQ3. Value for Adaptive Tasks :** How much redundant code can Client Insourcing eliminate by *replicating* only the necessary remote functionality? Are our centralized variants amenable to be *redistributed* with a third-party automated distribution tool? (Section 3.4.4)

[7]https://github.com/frictionlessdata/tableschema-py

- **RQ4. Value for Perfective Tasks :** How suitable are the centralized variants of distributed subjects for isolating and removing common performance bottlenecks? How much does Client Insourcing reduce the task complexity as compared to the original debugging process? (Section 3.4.5)

3.4.1 Evaluation Setup

To evaluate our approach, we have applied it to insource **61** different remote executions of **10** full-stack JavaScript applications. Table 3.1 summarizes the information about invoking these remote functionalities for each application. These remote services differ in their HTTP methods (e.g., GET, POST, PUT etc.), types of parameters, return results, and business logic.

To confirm that our approach is widely applicable, we selected as our evaluation subjects open-source full-stack JavaScript applications with dissimilar HTTP frameworks used to implement their client (Tier 1), server (Tier 2) and database (Tier 3) parts: **Tier1:** JQuery, Ajax, fetch, axios, AngularJS, and Angular2-TS; **Tier2:** Express, koa.js, and Restify, and **Tier3:** MySql, Postgres, and knex.js.

3.4.2 Saving Effort with Client Insourcing

Although developers can insource remote components by hand, the resulting program transformations can quickly become laborious and error-prone, especially for functionalities scattered across multiple script files and database-dependent code appearing in non-JavaScript files. Hence, the value of **JS-RCI** lies in automating the transformations required to insource these components. With **JS-RCI** completely automating the refactoring, the programmer would not have to modify any code by hand. To estimate the effort saved by **JS-RCI**, we

Table 3.1: Subject Distributed Apps and Client Insourcing Results

Subject Apps (Tier1,Tier2,Tier3)	**HTTP Methods**	**Remote Services**	*C&P/M* (ULOC)
recipebook (AngularJS↔Express ↔MySQL)	GET	/recipes	22/45
	GET/PUT/POST/DEL	/recipes:id	72/172
	POST	/ingredients	25/48
	GET/PUT/DEL	/ingredients:id	74/207
	POST	/directions	26/57
	GET/PUT/DEL	/directions:id	60/130
DonutShop (Ajax↔Express ↔knex)	GET/POST	/donuts	22/88
	GET/POST/DEL	/donuts:id	29/155
	GET/POST	/employee	20/71
	GET/POST/DEL	/employee:id	29/138
	GET/POST	/shops	16/83
	GET/DEL	/shops:id	19/128
res-postgresql (axios↔restify↔Postgres)	GET/POST	/user	22/71
	GET/PUT/DEL	/user	40/120
med-chem-rules (fetch↔koa.js↔knex)	GET	/hbone	9971/9994
	GET	/molecular	9974/9997
theBrownNode (JQuery↔Express)	GET	/users/search	37/65
	GET	/users/search/id	36/64
Bookworm (AngularJS↔Express)	GET	/api/ladywithpet	394/409
	GET	/api/thedea	394/409
	GET	/api/theredroom	394/409
	GET	/api/thegift	394/409
	GET	/api/wallpaper	394/409
	GET	/api/offshore	394/409
	GET	/api/bigtripup	394/409
	GET	/api/amont	394/409
realty_rest (Angular2↔Express)	GET	/properties	284/297
	GET	/properties:id	287/300
	GET	/brokers	86/99
	GET	/brokers:id	90/103
	GET/POST/DEL	/prprts/favs	34/73
	POST	/prprts/likes	291/304
ConferenceApp (Angular2↔Express)	GET	/findAllSpeakers	13/66
	GET	/findSpeakerById	15/68
	GET	/findAllSessions	43/117
	GET	/findSessionById	46/119
Employee Dir (Angular2↔Express)	GET	/employees	22/44
	GET	/employees/id	38/60
shopping-cart (Angular2↔Express)	GET/POST/DEL	/cart-items	79/130
Total	**61**		**24.9K/26.6K**

use the ULOC (Uncommented Lines of Code) that would have to be copied at the server and pasted to the client as well as the ULOC that would have to be modified at the client for each remote service. Thus, modified client code (**M**) includes the copied/pasted code (**C&P**). For the 61 remote services of 10 applications, **JS-RCI** eliminates the need to modify the client code as many as **26,685** ULOCs in total, **20,073** ULOCs are database code.

3.4.3 Correctness of Client Insourcing

The applicability of **JS-RCI** hinges on whether Client Insourcing preserves the execution semantics (i.e., business logic) of the refactored applications, a property we refer to as *correctness*. A subject application's original and refactored versions are expected to successfully pass the same test cases. Some of the tests that come with our subjects are also distributed, invoking server-side functionalities through HTTP middleware. To use their remote parameters and results as test invariants, we manually transformed these tests for local execution without middleware. Altogether we ran 61 test cases against the original and insourced versions of our subject applications, with all of them successfully passing. It is possible that for some complex or esoteric cases, the correctness of Client Insourcing would not be as stellar, but by examining why a test case failed, the programmer can always correct the insourced code.

The Effectiveness and Correctness of Detecting the Marshalling Points

Recall that in Section 3.3, we proposed two search strategies—*Idempotent Execution* and *Fuzzing*—to detect the marshalling points of a refactored application. To compare and contrast the effectiveness and correctness of these strategies, we ran our analysis procedure with each of these strategies in isolation.

We observed that Idempotent Execution with its Record/Replay phases removes the *false-negatives* in the detected marshalling points for stateful servers. Our results show that subject applications with only safe (or read-only) operations are not affected by the restoring process (20/61). However, we discovered that idempotent execution is critical for the majority of our subjects (41/61). Specifically, having been changed by HTTP PUT/POST/DELETE requests, global variables were restored correctly in realty-rest and database entries were restored in other subjects.

In contrast, Fuzzing removes *false-positives* for detecting marshalling points. We discovered that Fuzzing proved effective also in twelve cases of our subjects (12/61). Hence, to infer the correct set of marshalling points, while removing both false-negatives and positives, **JS-RCI** applies both strategies in turn.

Table 3.2: Correctness affected by Search Strategies

Subject Apps	Stateless	DB	All	w/o Fuzzing	w/o Idem_Ex
theBrownNode	✓	X	2/2	0/2	2/2
Bookworm	✓	X	8/8	0/8	8/8
ConferenceApp	✓	X	4/4	4/4	4/4
EmployeeDir	✓	X	2/2	2/2	2/2
shopping-cart	X	X	3/3	3/3	0/3
realty-rest	X	X	8/8	6/8	2/8
recipebook	X	✓	13/13	13/13	0/13
DonutShop	X	✓	14/14	14/14	0/14
res-postgresql	X	✓	5/5	5/5	0/5
med-chem-rules	✓	✓	2/2	2/2	2/2
Total			**100%(61/61)**	**80%(49/61)**	**32%(20/61)**

3.4.4 Insourcing's Value for Adaptive Tasks

Value of Automated Enabling of Disconnected Operation

In lieu of Client Insourcing, developers would have to replicate remote functionalities by hand. Unassisted by program analysis, a programmer remains unaware which specific code entities comprise a remote functionality that needs to be replicated. Hence, a safe option for manually replicating any non-trivial remote functionality would be to first duplicate the entire server-side source file at the client, and then adapt the duplicated code as necessary. Notice that such copy-and-modify procedures invariably introduce some unnecessary code, which is never used but still needs to be deployed and maintained. Hence, in our evaluation, we count the number of lines of such unnecessary code that could result from copying the entire source file from the server to the client.

Table 3.3: Replication for disconnected operations

Subject Apps	S_{LOC}	S^{CI}_{LOC}	S_{LOC} - S^{CI}_{LOC} (Unnecessary LOC)
theBrownNode	120	76	44
Bookworm	340	299	41
realty-rest	457	420	37
ConferenceApp	78	51	27
EmployeeDir	56	35	21
shopping-cart	48	26	22
recipebook	624	376	126
DonutShop	455	308	147
res-postgresql	73	28	45
med-chem-rule	10228	9976	252

To identify the code portions that are indeed unnecessary to replicate the remote functionalities under consideration, we first count the total lines of JavaScript code taken to implement the original server parts of each subject app (S_{LOC}). To replicate all remote functionalities, programmers would copy S_{LOC} to the client and adapt them as necessary. The

copied S_{LOC} are intermingled with various unnecessary parts, including middleware, fault handling, or no-longer relevant comments. The values of S_{LOC} are computed by examining the programmer-written files and their dependencies deplofyed in the Node.js server. In contrast, Client Insourcing extracts from the server only the lines of code required to implement the replication disconnected operation (S^{CI}_{LOC}). For simplicity, we assume that the entire remote functionality is replicated for each subject application. To estimate the number of lines of code that Client Insourcing saves from being replicated unnecessarily, we subtract S_{LOC} from S^{CI}_{LOC} as shown in Table 3.3.

Value of Centralized Variants for Redistribution

Client Insourcing creates a redistributable (centralized) application variant that can be *refactored* and *enhanced* using any state-of-the-practice program transformation tools and then *distributed anew* using any state-of-the-art ditribution tools. We applied two JavaScript refactoring tools on our centralized variants: NODE-SANDBOX[8] for security enhancements and EXTREMEJS [105] for redistribution. NODE-SANDBOX prevents untrusted JavaScript code from executing infinite loops or consuming large volumes of heap memory in the isolated code. However, sanboxing frameworks incur a heavy performance penalty on the isolated code, and as such must be used sparingly, if the application is to remain usable. Hence, the code to sandbox is typically isolated from the rest of the application to run in its own process and address space. EXTREMEJS automatically distributes centralized JavaScript applications at the function level of granularity.

$$App_{dist} \rightarrow \text{JS-RCI} \xrightarrow{App_{cent}} \text{NODE-SANDBOX} \xrightarrow{App^{sanboxed}_{cent}}$$
$$\text{EXTREMEJS} \xrightarrow[\text{Client Stub}]{\text{Remote Stub}} App^{sanboxed}_{dist}$$

[8]Node-SandBox (https://github.com/patriksimek/vm2)

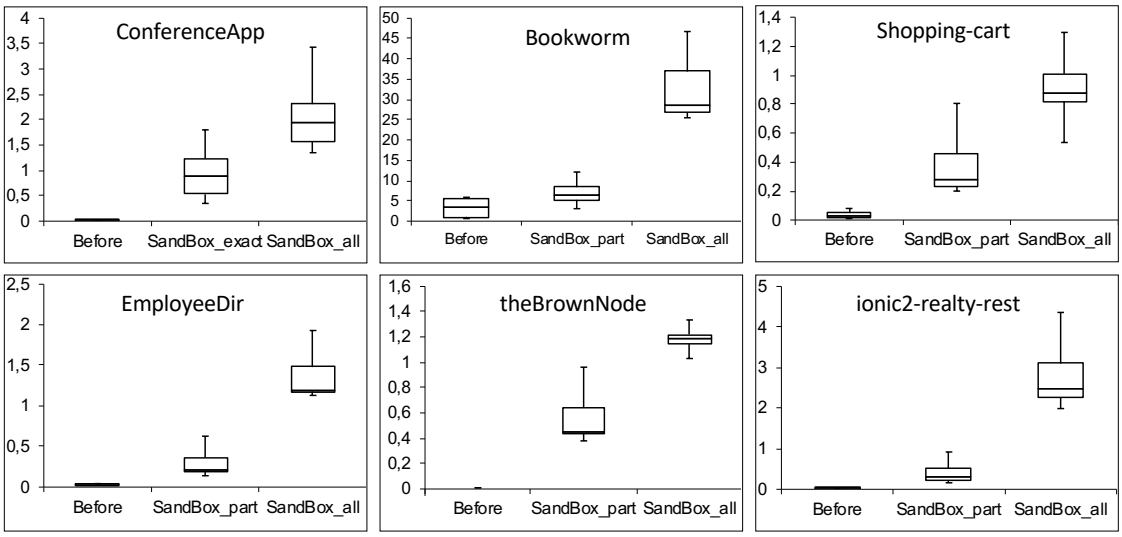


Figure 3.6: Redistribution with Sandboxing

In our evaluation, we measure the additional execution time incurred by sandboxing only a subset of the remote functionality vs. the entire original remote functionality. This comparison highlights the importance of isolating only the code that needs to be sandboxed. Figure 3.6 shows by how much sandboxing increases the execution time for two versions of the subject applications: (1) only the needed subset of the server part is isolated (*SandBox_part*); (2) the entire server part is isolated (*SandBox_all*). The observed differences in execution time between these two versions are quite striking, clearly showing that sandboxing the entire server part is impractical.

3.4.5 Insourcing's Value for Perfective Tasks

Consider the problem of identifying the source of a performance inefficiency or bottleneck in a distributed app. First, one has to be able to exclude the reasons of misconfiguration or network volatility among the potential causes. Then, one has to make sure the app is free of known architectural anti-patterns [86]. For example, consecutive fine-grained remote invocations can be batched to take advantage of better progress being made in increasing

the bandwidth as compared to the latency characteristics of modern networks [81]. However, the sources of inefficiency can be more subtle than those stemming from ill-conceived architectural decisions. At some point, the debugging focus may need to switch to the programmer-written code. The JavaScript ecosystem features numerous libraries, so the same functionality can be implemented in a variety of ways, each of which may have its own performance characteristics. Choosing one programming idiom over another can have a dramatic effect on the overall app performance [37, 38]. Given the divergent performance characteristics of different JavaScript APIs, several prior work directions have focused on identifying and removing common sources of inefficiency. The approach presented in [87] empirically identifies recurring patterns of inefficient program performance, so they can be restructured, thereby improving the overall performance. That kind of restrucuring is a common example of perfective modifications. However, the majority of the state-of-the-art approaches that identify and remove performance inefficiencies target centralized programs. Client Insourcing can make these approaches applicable to distributed apps.

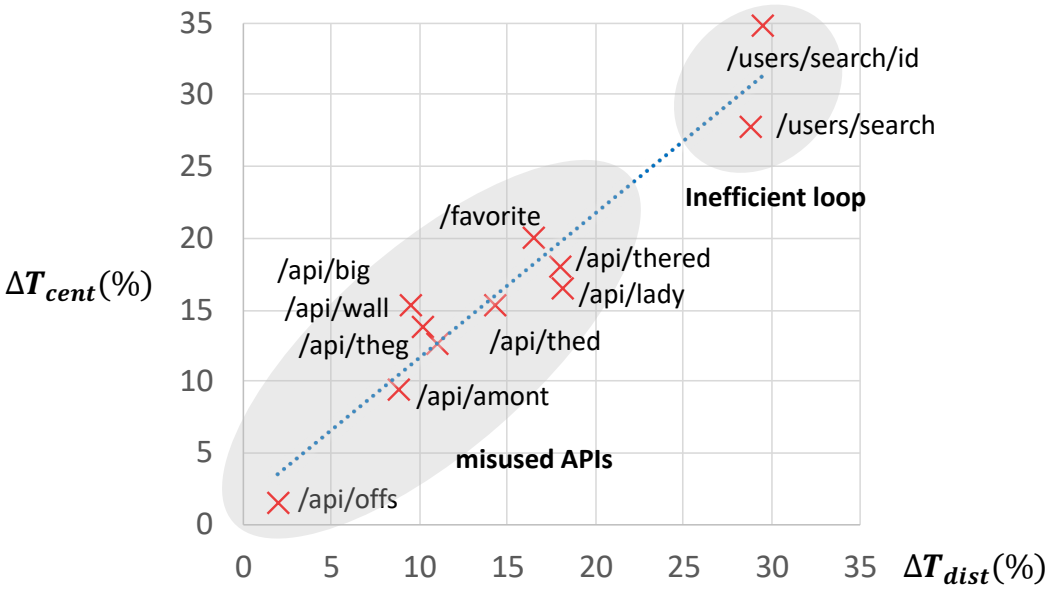


Figure 3.7: Scatter Plot and Regression Test for ΔT_{dist} versus ΔT_{cent}

We applied the approach presented in [87] to the centralized variants of our subject apps produced by means of Client Insourcing. Out of 61 subjects, 11 ended up containing some known patterns of performance inefficiency. For example, *Bookworm* repetitively misused unoptimized string API patterns: `data.split("...").join("asdf").split(".").join("asdf")`. By taking the network and middleware functionality out of the list of suspected causes of performance problems, Client Insourcing enables so-called "isolated profiling," which isolates the programmer-written code to be used as the sole target of analysis and optimization efforts. To demonstrate the value of Client Insourcing, we removed all the pointed-out 11 performance bottlenecks from both the original subjects and their centralized variants. As it turns out, the bottleneck removals improved the performance of both versions (distributed and centralized) of each subject. Figure 3.7 summarizes the observed performance improvements. For the original distributed subjects (ΔT_{dist}), the improvements range between 29.5% and 2.0%. For their centralized variants (ΔT_{cent}), the improvements range between 34.8% and 1.6%. We also applied a linear regression analysis to compute how closely ΔT_{dist} and ΔT_{cent} correlate with each other, resulting in $\Delta T_{cent} = \mathbf{1.0089} * \Delta T_{dist} + \mathbf{1.556}$. This equation shows that ΔT_{cent} and ΔT_{dist} are almost perfectly correlated, so centralized variants can indeed serve as reliable and convenient proxies for an important class of performance debugging and optimization tasks.

In addition, Client Insourcing reduces the complexity of the debugging process by streamlining the debugged subject's execution flow: from the complexity of distributed execution over the Web to the simplicity of centralized execution. To quantify the actual value of debugging the centralized variant of a web application instead of its original distributed version, we compared the total execution time taken by invoking distributed functionalities vs. their local insourced counterparts. We assumed that the debugging task was identifying performance bottlenecks, so we heavily instrumented our benchmarks before measuring

their execution performance. As it turns out, insourcing reduces a distributed functionality's execution time by more than **90%** on average. Given that debugging typically involves repeated executions, having much faster subjects to debug should improve the efficiency of the debugging process.

3.4.6 Threats to Validity

The validity of our evaluation results is subject to both internal and external threats that we discuss in turn next.

Internal Threats

One of our evaluation criteria is the performance of the JavaScript code generated by our implementation of the Client Insourcing refactoring. The performance of JavaScript code is known to be heavily affected by specific design and implementation choices. Similarly, our own JavaScript coding practices are likely to have affected the observed performance characteristics. For example, rather than directly inject the insourced code segments into the client source files, we choose to create brand new source files for each insourced languages declaration, with the new files simply included in the original files. Client Insourcing could have been implemented in a variety of other ways, possibly yielding different software engineering and performance metrics.

External Threats

All our performance measurements were performed on (DELL-OPTIPLEX5050, running the JavaScript V8 Engine (v 6.11.2). Due to the popularity of JavaScript, the issue of maximizing the efficiency of JavaScript engines has come to the forefront of system design [87]. Although

V8 is a state-of-the-art JavaScript engine, it has its competitors, such as SpiderMonkey. Hence, the absolute performance of our experiments could differ if our measurements were run in a different execution environment.

3.5 Discussion

Our approach works only with relational databases interfaced with by means of SQL queries. Some non-SQL databases, such as MongoDB, use a distinct syntax in its client API. It should be possible to support the dissimilar CRUD operations of non-SQL databases, and we plan to explore such support as a future work direction.

For various reasons, some remote functionalities cannot be insourced to run on the client, thus making it impossible to create a centralized variant of certain distributed applications. In those cases in which distribution is inevitable, some application resources, naturally remote to the rest of the functionality, cannot change their locality. For instance, news readers display the stories deposited to some centralized repository. It would be impossible to move the news functionality away from the repository to the client, without manually creating some mock components that realistically emulate the appearance of news content locally. In other words, some remote functionalities may depend on resources that cannot be easily migrated away from their host environment for reasons that include relying on server-specific APIs or being dependent on some hard-to-move infrastructure components.

In addition to standard commands, HTTP also provides a separate WebSocket interface that opens a dedicated TCP/UDP connection after a round-trip handshake. WebSocket-based communication is fundamentally asynchronous and is used mostly in streaming scenarios. Although Client Insourcing can also help in the re-engineering of web apps that use WebSocket for non-streaming scenarios, we left the support for this part as a future work direction.

Some web applications may span across more than two tiers. Our reference implementation assumes a two-tier client-server application with a possible server-side SQL database, in which both tiers are implemented in JavaScript. It should be possible to extend Client Insourcing to multi-tier applications, perhaps by applying the two-tier technique pairwise to each respective pair of tiers. At the same time, flattening tiers may not work well for mobile execution environments, which are known to be resource-scarce.

Chapter 4

A Debugging Approach for Web Applications

Most programmers abhor debugging, due to its arduous, wasteful, and tedious nature. It can be much harder to debug distributed applications than centralized ones. Distributed systems suffer from partial failure, in which each constituent distributed component can fail independently. In addition, non-trivial bugs, including performance bottlenecks and memory leaks, can be caused by server/middleware misconfigurations or intermittent network connectivity rather than by any problems in the programmer-written code. Programmers need novel debugging approaches that can pinpoint whether the cause of a non-trivial bug in a distributed application is indeed in the programmer-written code.

To alleviate the challenges of debugging distributed applications, we present a novel debugging approach that takes advantage of automated refactoring to remove much of the uncertainty of distributed execution from the debugged programs. Our approach first transforms a distributed application into its semantically equivalent centralized version by applying our domain-specific refactoring, *Client Insourcing*, which automatically moves a server-based remote functionality to the client, replacing middleware communication with local function calls. *Client Insourcing* is a refactoring, as the resulting centralized application retains its execution semantics. Then standard debugging techniques are applied to debug this centralized application. After the bug is localized and fixed, our approach generates a patch that

is applied to the faulty part of the distributed application. We call our approach *Catch & Release* or CANDOR for short, as it *catches* bugs in the centralized version of a distributed application, and after fixing the bugs, *releases* the application for its continued distributed execution.

We implement CANDOR for the important domain of full-stack JavaScript applications, in which both the client and server parts are written and maintained in JavaScript, and evaluate its effectiveness in fixing two important types of bugs known to be prevalent in this domain: memory leaks and performance bottlenecks. Our evaluation applies our approach to localize and fix bugs that were previously found in third-party applications. We verify the correctness and value of our approach by applying our bug-fixing patches to the faulty versions of these applications and then confirming that the patched versions pass the provided test suites. We argue that CANDOR reduces the complexity of the debugging process required to fix these bugs by reporting on our experiences.

This paper makes the following contributions:

1. We present a novel debugging approach for distributed applications that uses automated refactoring to produce a semantically equivalent, centralized versions of the debugged subjects. Any of the existing state-of-the-art debugging techniques become applicable to track and localize bugs in such centralized versions. (CATCH)

2. We develop automated bug patching, which given the bug-fixing changes of the debugged application's centralized version, replays these changes on the application's client or server parts. (RELEASE)

3. We empirically evaluate the correctness and value of our approach by applying it to track and localize known bugs in real-world third-party full-stack JavaScript applications.

```
1  //server part in theBrownNode
2  var express = require('express'), app =
       express.createServer(..);
3  var users=[]; ...
4  app.post('/users/search', function(req,
       res) {
5   var data = req.body; //client-input
6   var result=getUsers(data);//serv-output
7   res.send(result);});
8  function getUsers(searchUser) {
9     return getObjsInArray(searchUser,
          users);}

1  //inefficient for-in loop in server part,
       lines are from 5 to 18
2  function getObjsInArray(obj, array) {
3   var foundObjs = [];
4    for (var i=0; i < array.length; i++){
5     for(var prop in obj) {
6      if(obj.hasOwnProperty(prop)) {
7       if (obj[prop] === array[i][prop]) {
8        foundObjs.push(array[i]);
9        break;
10    }}}}
11   return foundObjs;
12 }
```

```
1  //client part in the BrownNode
2   $.ajax({
3   url: '/users/search',
4   data: {fName: $('#fName').val(),
5      ...},//client-input
6   type: 'POST',
7   success: function(data) {//serv-output
8   $('#results').text(JSON.stringify(data));}});

1  //patch for server part
2
3  5,18c1,16
4  <(original code for getObjsInArray)
5  ---
6  > function getObjsInArray(obj, array) {
7  >  var foundObjs = [];
8  >  var keys = Object.keys(obj);
9  >  for (var i=0; i < array.length; i++) {
10 >   for (var j = 0, l = keys.length; j < l;
       j++) {
11 >    var key = keys[j];
12 >    if (obj[key] === array[i][key]) {
13 >     foundObjs.push(array[i]);
14 >     break;
15 >  }}}
16 >  return foundObjs;
17 > }
```

Figure 4.1: Distributed App *theBrownNode* and patch for an inefficient iteration

4.1 Debugging Full-Stack JavaScript Applications

In this section, we explain our approach to debugging distributed full-stack JavaScript applications by discussing how it facilitates the process of locating bugs in two real-world examples.

4.1.1 Motivating Example I: *Removing Performance Bottlenecks*

Consider the code snippet in Fig. 4.1, in which the remote service `/users/search` of the distributed app *theBrownNode* calls function `getUsers`, which contains nested `for` loops. The client portion invokes the server-side script `/users/search`, passing various query parameter data to obtain the search query results. The code of the inner loop is quite inefficient, as it performs two conditional checks. Being on a hot execution path, this inefficiency causes a noticeable performance degradation. One can remove this bottleneck by eliminating the need to check whether the property `prop` is indeed defined in the object `searchUser` and not inherited from `searchUser` prototype: to exclude the inherited properties, the code can be optimized to use `Object.keys()` [87].

Notice that in the original distributed version of this application, it would be non-trivial to locate the actual source of this performance bottleneck. The performance of a distributed application can be affected by myriad factors, many of which have nothing to do with the application's implementation. To meet the expected performance requirements, servers must be properly configured for the actual usage scenarios, and so is the middleware infrastructure that encapsulates the communication functionality between the client and server components. In addition, network connectivity and utilization can affect the overall performance. Intermittent network connectivity and bandwidth saturation can lead to uncertain periods of poor network performance.

Even if the programmer were to become certain that the cause of the observed performance bottleneck lies in the implementation, localizing the source location of the bug in a distributed application can be a complex undertaking that requires generating a large volume of synthetic HTTP traffic against a specially instrumented version of the server. Then the client parameters would have to be matched against the resulting server execution profiles.

This debugging procedure is complicated, as it requires a customized server deployment and the examining of the remotely generated performance profiles.

With CandoR, the programmer first replaces the remote invocation of `/users/search` with an equivalent local function call, thus eliminating all middleware functionality and server-side execution. Once the remote code is insourced, the resulting centralized program can be easily debugged by using any existing tools for JavaScript programs. Rather than transferring log files from the server to the client and trying to correlate different remote executions with their parameters, the programmer can debug the execution of local function `users_search`. Once the programmer changes the insourced version to fix the bug, CandoR automatically generates a fix patch (the shaded code snippet in Fig. 4.1) to be executed against the original server or client part of the distributed application (i.e., the "release" phase).

4.1.2 Motivating Example II: *Detecting Memory Leak*

Some of the most common bugs afflicting remote services are memory leaks. Consider function `leakingService` in Figure 4.2 that represents a simplified server-side service invoked by various remote clients. These clients invoke the service by means of distribution middleware that hides all the low-level details on the client-server communication. Notice that every time this function completes its execution, it leaks some memory, as random String is appended to the globally declared Array `leak`, which is never garbage-collected. Although this example is simplified for ease of exposition, it is representative of numerous anti-patterns that can quickly exhaust the server's memory upon heavy utilization.

This bug is also quite challenging to detect and fix. One first has to be certain that the memory leak in question is not due to server/middleware configuration problems. In addition, the very presence of middleware functionality makes it hard to locate memory bugs in

```
1  // every time this service is invoked,
2  // it "leaks" a bit of memory, as
3  // var leak is never garbage-collected
4  var leak = [];
5  function leakingService() {
6   leak.push(Math.random()+" on a stick,
        short!");
7  }
8  http.createServer(function (req, res) {
9     leakingService();
10    res.end("success");
11 }).listen(1337);
```

```
1  // invoking leakingService in client
2  let data = '';
3  http.get(S_URL, (res) => {
4     res.on('data', (chunk) => {
5     data = JSON.parse(chunk);
6    });...);
```

```
1  //patch for server part
2  16,18c1,19
3  <(original code for leakingService)
4  ---
5  >(definition of delegating writeToFile)
6  > function leakingService() {
7  >  writeToFile(Math.random()+" on a stick,
       short!");
8  > } //leakage detected in var leak
```

Figure 4.2: Memory Leak Examples for Server and Client parts

the programmer-written code. Much of the client/server distributed execution flows through middleware libraries, whose memory consumption and footprint can conceal the actual locations in the programmer-written code that contain memory-related bugs.

To help developers test the remote functionality, the Node.js framework provides testing libraries, using which one can script HTTP requests against a given server. These libraries help verify whether the input and output values of a service being tested are as expected. These functional testing utilities cannot help identify whether the server code is leaky, however.

In the absence of fully automated techniques for debugging *Full-Stack JavaScript Applications*, developers have no choice but to manually instrument both the client and the server parts of the debugged applications. More specifically, the current state of the art in detecting memory leaks in JavaScript programs involves taking and comparing with each other multiple heap snapshots in the suspect regions of the server-side functionality. A commonly used technique for finding memory leaks in web applications is *three snapshots* [103]. Even

detecting a sufficient degradation in performance of the server-side functionality requires some client to execute multiple consecutive HTTP requests. As a result, to reproduce a memory leak bug, programmers are expected to follow a complex and tedious debugging process.

In contrast, CandoR takes a drastically different approach to debugging full-stack JavaScript applications. It performs all bug localization tasks on the distributed application's centralized version, in which both the client and server parts execute within the same JavaScript interpreter. This centralized version is generated automatically via a new refactoring that we call *Client Insourcing*. This refactoring moves the server-side functionality to the client, so it can be invoked by calling local functions rather than through the layers of distribution middleware such as HTTP Client. In essence, Client Insourcing integrates the remote, potentially buggy functionalities with the client code, so all the debugging techniques for centralized JavaScript applications can be applied to the insourced application. For example, state-of-the-art modern JavaScript execution environments provide built-in profiling infrastructures that can be applied to any running application. A centralized application can be re-executed at will without having to coordinate the execution of multiple remote execution nodes. Because Client Insourcing replaces all distributed functionality with direct local function calls, the identified memory leaks would indeed stem from the programmer-written code rather than any server/middleware misconfiguration.

4.2 CandoR: Design & Reference Implementation

CandoR works in three phases. First, the server part is automatically insourced, producing a centralized application whose semantics is equivalent to the original distributed full-stack JavaScript application. The resulting centralized application is then debugged by means of

any of the existing techniques for locating and fixing bugs in JavaScript programs. Finally, based on the before (i.e., buggy) and after (i.e., fixed) versions of the centralized application, CANDOR generates a patch to be executed against the application's original client or server parts, thereby applying the fix to the correct portion of the distributed application.

4.2.1 The Client Insourcing Automated Refactoring

Full-stack JavaScript applications comprise client-side and server-side JavaScript code. The *Client Insourcing* automated refactoring first identifies the remotely invoked functionalities of the server code by statically analyzing the corresponding marshaling points of the parameters passed by the client to the server and the server's output to the client (i.e., marked as `//client-input` and `//serv-output` parts respectively in Fig. 4.1). The process requires no manual steering from the programmer, whose role is limited to running the application's test suites under standard input and transferring the generated log file of the marshaling points to the server. Parameterized with this file, dynamic symbolic execution then computes a transitive closure of the server-side statements executed by the remote invocations. Client Insourcing analyzes JavaScript programs by using the z3 SMT solver [24], similarly to other declarative program analysis frameworks[62, 98].

The computed relevant server statements are then insourced into the application's client part. The insourced statements are placed in automatically generated client-side functions. These functions are invoked directly without any middleware. So the refactoring process completes by replacing all middleware-based invocations with direct calls to these functions (see the equivalent centralized version of *theBrownNode* in Fig. 4.3). This refactoring preserves the application's business logic, while significantly simplifying its control flow. Rather than spanning across two JavaScript engines (client and server), the resulting centralized applications

```
//equivalent Centralized App
var users=[]; //insourced
function getObjsInArray(obj,
     array){...} //Insourced(
     buggy iterations)
function getUsers(searchUser)
     {...} //Insourced

function users_search(i){
 var out=getUsers(i);
 return out;
} //Insourced

//$.ajax({url: '/users/search
     ',...}) is replaced by
var input = {fName:...};
var output=users_search(input);
$('#results').text(JSON.
     stringify(output));
```

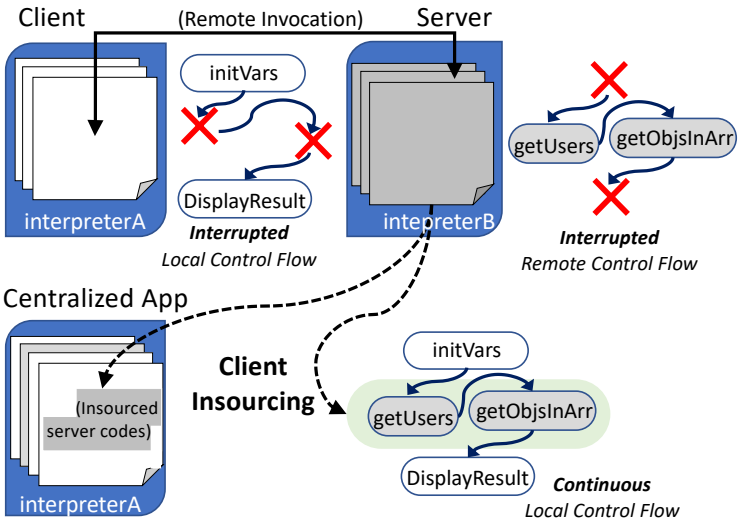


Figure 4.3: Continuous Control Flow of Distributed Codes constructed by Client Insourcing

require only one engine. Since JavaScript engines often differ in terms of their debugging facilities (e.g., logging support, information messages, etc.), interacting with only one engine reduces the cognitive load of debugging tasks. In addition, one of the key hindrances that stand on the way of debugging distributed applications is the necessity to keep track when the control flow changes execution sites. The control flow of a full-stack JavaScript application can go through any of the constituent application parts: client, server, and middleware. Ascertaining when the flow crosses the boundaries between these parts can be challenging, particularly if the maintenance programmer, in charge of a debugging task, is not the same programmer who wrote the original buggy code. By transforming the original application into its centralized counterpart, Client Insourcing creates a debugging subject with a regular local control flow that is easy to follow with standard debugging tools (Fig. 4.3).

4.2.2 Catching and Fixing Bugs in Insourced Apps

Insourcing produces centralized applications that can be debugged by means of any of the existing or future JavaScript debugging techniques. CANDOR makes all these state-of-the-art debugging techniques immediately applicable to full-stack JavaScript applications. Automatically produced equivalent centralized versions are easier to execute, trace, and debug, due to their execution within a single JavaScript engine. Next, we explain how CANDOR can help remove performance bottlenecks and memory leaks.

Identifying and Removing Performance Bottlenecks The interpreted, scripting features of JavaScript make the language a great fit for rapid prototyping tasks. Unfortunately, deadline pressures often leads to programmers having to move such prototyped code into production. Once deployed in actual execution environments, this code frequently suffers from performance problems. Several previous works address the challenges of uncovering non-trivial recurring cases of performance degradation in JavaScript applications [37, 48, 87]. For example, reference [87] identifies 10 common recurring optimization patterns: 2 inefficient iterations, 6 misused JavaScript APIs, and 2 inefficient type checks. One can find these patterns statically by analyzing a JavaScript codebase. Notice that static analysis can be applied separately to the client and server parts of a full-stack JavaScript application. However, applying the *Pareto Principle* [111] to program optimization, one can expect a typical program to spend 90% of its execution time in only 10% of its code. Hence, to verify whether the found inefficiencies are indeed the sources of performance bottlenecks requires dynamic analysis, which is much easier to perform on the centralized version of a debugged distributed application. Specifically, the centralized version is instrumented and its runtime performance profile is generated. Then each candidate inefficiency is removed in turn and another profile is generated. By comparing the original profile and that of a modified version, one can verify whether the latest fix was indeed for a performance bottleneck-causing

bug. Without a centralized version, the number of performance profiles would need to at least double, and the server part would require a separate execution driver to generate its profiles.

Fixing Memory Leaks When fixing memory leaks, programmers typically store the execution traces of leaky code persistently for a subsequent examination. When debugging real-world web applications, programmers often can delegate the logging task to a third-party service. However, to fix a memory leak in a distributed version, both the client and server parts need to be logged. In contrast, with CANDOR, programmers can localize memory leaks by applying a memory profiler such as memwatch[1] to the debugged application's centralized version. As shown in the Fig. 4.2, memwatch detects the leaking global array `leak` in the centralized version, with the fix replacing `leak.push` with `writeToFile`[2]. CANDOR then generates a patch to be applied to the application's server part.

4.2.3 Releasing the Bug Fixes

Once the programmer fixes the bug in the application's centralized version, the resulting fixes have to be applied to the actual client and server parts of the original application, thus completing the final *release* phase of the CANDOR debugging process. To that end, CANDOR automatically generates input scripts for GNU Diffutils[3], which executes these scripts against the source files of the original full-stack JavaScript application by using GNU patch[4].

To correctly generate a diff script that modifies the affected lines of the original applications, CANDOR keeps track of the correspondences between the application's original and insourced versions. This process is complicated by the multi-step nature of Client Insourcing transfor-

[1] https://github.com/eduardbcom/node-memwatch
[2] For additional implementation details, see https://bit.ly/2B9a3wf
[3] https://www.gnu.org/software/diffutils
[4] http://savannah.gnu.org/projects/patch

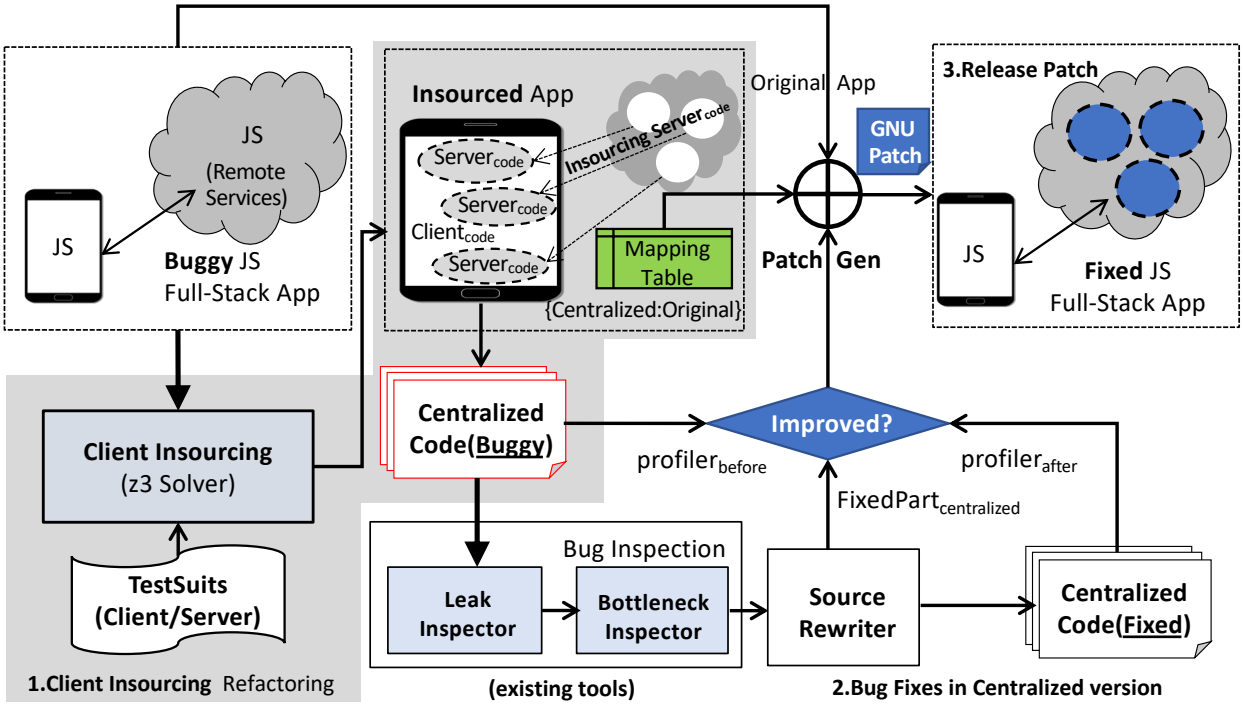


Figure 4.4: Debugging Full-Stack JavaScript Applications with CANDOR

mations. Because the basic insourcing unit is a function, all free-standing server statements are first placed into individual functions, through a process that synthesizes new function names and applies the *extract function* refactoring on the free-standing statements. We call this process *normalization*. The actual insourcing transformation is applied at the function level of granularity.

CANDOR keeps track of how the lines map between the original client and server source files and their centralized version. This mapping is used to automatically generate a patch that replays the bug fixing changes of the centralized version on the original source code's client or server portions (Fig. 4.4).

4.3 Evaluation

- **RQ1—Correctness:** Does Client Insourcing preserve the execution semantics of full-stack JavaScript applications? Are existing test-suits still applicable to the centralized variants of the debugged applications? (**4.3.1**)

- **RQ2—Value:** By how much does CandoR reduce the debugging complexity in terms of the number of steps and tools required to localize and fix bugs? (**4.3.2**) How much programmer effort can CandoR save? (**5.3.2**)

4.3.1 Evaluating the Correctness of Client Insourcing

Table 4.1 shows subject full-stack applications and their remote services. The size of each subject application is shown in terms of the number of uncommented lines of JavaScript code (ULOC) for the server (S_{ULOC}) and the client (C_{ULOC}) parts. Client Insourcing changes the architecture of full-stack JavaScript applications from distributed to centralized by combining their server and client parts. CI_{ULOC} indicates ULOC for the centralized version of each subject.

The applicability of CandoR hinges on whether Client Insourcing preserves the execution semantics (i.e., business logic) of the refactored applications, a property we refer to as *correctness*. In modern software development practices, applications are maintained alongside their test suites, a collection of test cases that verify each important unit of application functionality. In our correctness evaluation, we leverage the ready availability of such test suites for our subject applications. In other words, the original and refactored versions of a subject application is expected to successfully pass the same set of test cases.

Some tests in the available test suits are also distributed, in that they invoke remote services

by means of HTTP client middleware, which marshals input parameters and unmarshals returned values. It is the returned values that are used as test invariants. We had to manually transform such distribution-specific tests to work against the centralized (insourced) versions of their test subjects.

Table 4.1: Subject Distributed Apps and Client Insourcing Results

Subject Apps	S_{ULOC}	C_{ULOC}	Remote Services	CI_{ULOC}
theBrownNode	147	43	/users/search /users/search/id	37 36
Bookworm	371	1814	/api/ladywithpet /api/thedea /api/theredroom /api/thegift	394 394 394 394
search_leak	34	13	/search_leak	17
ionic2_realty_rest	453	387	/properties/favorites	24

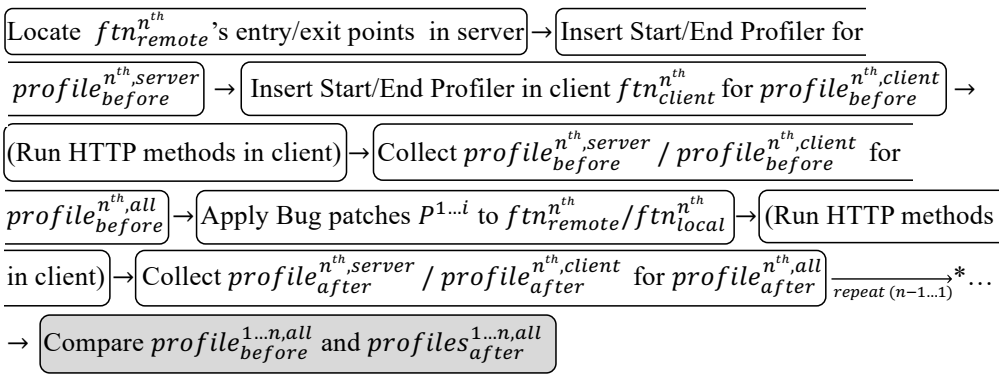
Table 4.1 shows the total number of tests in each evaluated test suite, including the number of tests manually transformed to work against the centralized versions of subject applications; the table shows whether tests successfully passed in the original and refactored version of each subject. Based on these results, we can conclude that Client Insourcing shows a high degree of correctness($\frac{8}{8} \cdot 100{=}100(\%)$), as the same of number of successful tests is passed by the refactored applications, making them suitable for debugging.

4.3.2 Case Study: Traditional vs. CANDOR-Enabled Debugging

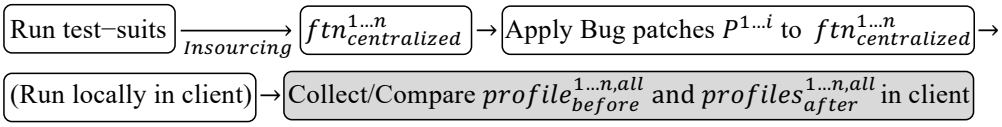
In this case study, we compare and contrast a traditional approach to localizing a bug in a full-stack JavaScript application and the **CANDOR** debugging approach. In this case study, we assume that a programmer needs to debug a distributed application with n remote functionalities[5] $ftn^{1...n}_{remote}$ to produce i corrective patches $P^{1...i}$; applying the patches fixes the found bugs. We assume that standard profiling is used to stamp the start and the end of executing

[5]Each remote functionality is exposed as a remote service invoked via some middleware API.

each remote service, so as to obtain the total execution time and memory footprint. To the best of our knowledge, no automated tools can identify the entry/exit points of a server-side remote functionality invoked by clients. Hence, the programmer is expected to manually examine the server-side code to locate and instrument these entry and exit points for every remote functionality in question. In some cases, in order to instrument some business logic functionality, it must first be disentangled from any middleware-specific functionality. However, for ease of exposition, we disregard this additional required debugging-related task. Once the instrumentation is in place, a typical debugging procedure involves continuously invoking client-side HTTP requests against the instrumented remote functionalities. After a certain number of requests, the server-side logs then can be collected, transferred to the client, and examined for the obtained execution time and memory footprint numbers profiles (Fig. 4.5a).



(a) Typical Debugging Process for Full-Stack JavaScript Apps



(b) Streamlined Debugging Process with CandoR

Figure 4.5: Comparing the Debugging Processes

In essence, our approach reduces the accidental complexity of debugging; the essential complexity cannot be reduced, so localizing and fixing bugs will always remain a delicate and

complex task. Nevertheless, our approach allows programmers to focus on the actual debugging activities, unencumbered by the complexity of having to trace the execution of a buggy application across distributed sites. CANDOR simplifies the process by automatically identifying n remote functionalities and transforming them into equivalent n centralized local functions $ftn^{1...n}_{centralized}$, integrated with the client code. Afterwards, all the relevant debugging procedures can be applied to the resulting centralized application. Since these procedures are strictly local, they can be repeated at will, with their output examined in place. As a result, the number of debugging procedures decreases as compared to the traditional process, as shown in Fig. 4.5b.

4.3.3 Quantifying the Debugging Effort Saved by CANDOR

We see the main draw of CANDOR in that it reduces the amount of effort required to debug distributed applications to approximately that required to debug centralized ones. Although any debugging task can be cognitively taxing, tedious, and laborious, removing the complexity of distributed communication is expected to reduce these burdens. However, to be able to perform all debugging-related changes on the centralized version of a distributed application, these changes must affect the performance and memory consumption of both the distributed and centralized versions in comparable ways. In other words, if a change to the centralized version improves its performance or memory consumption, a similar improvement should be expected in the distributed version.

To check this hypothesis, we fixed different types of bugs in the centralized versions of 8 subjects, measuring their before and after execution time and memory consumption numbers. We then obtained the same measurements for their original and fixed distributed versions. Table 4.2 presents the performance and memory consumption improvements for these debug-

ging subjects. To measure performance, we use the V8 profiler. To reduce noise, we repeated each use case 2000 times and compared the average observed time elapsed: P_{before} and P_{after}, with the performance improvement calculated as $P_{improved} = \frac{P_{before}}{P_{before} - P_{after}} \cdot 100(\%)$. For the memory leakage bugs, we compared how much memory was used before and after the bug fixes by repeatedly executing the subjects 2000 times. The table's last column ($P^{D}_{improved}$ and $P^{CI}_{improved}$) shows the resulting percentage improvements for the distributed and centralized versions. As one can see, the improvement percentages are very close to each other, confirming that the centralized version can serve as a viable debugging proxy for its distributed application.

We also approximate the debugging effort saved by counting the number of uncommented lines of code (ULOC) that need to be examined by hand to successfully perform a debugging task. A successfully executed debugging task involves two phases: (1) localize the source line of the bug, (2) fix the bug by modifying the source code (i.e., generate a fix patch). In traditional debugging, phase 1 requires that all the executed client and server statements be examined, while with CandoR, Client Insourcing puts all the server statements executed by remote services into regular local functions (CI_{ULOC} in Table 4.1), thus eliminating the need to examine any remotely executed code to localize bugs. In phase 2, the bugs are fixed by applying automatically generated patches (F_{ULOC}).

4.3.4 Threats to validity

When implementing the patch generation module of CandoR, we made several design choices that may affect our evaluation results. For example, we measured the performance improvement of subjects running on a specific V8 Engine(v 6.11.2) and instrumenting machine(Dell-OptiPlex5050). However, the actual amount of improvement can change based

Table 4.2: Quantifying Debugged results by CANDOR

Remote Services	bug types [87]	F_{ULOC}	P^{D}_{before} (P^{CI}_{before})	P^{D}_{after} (P^{CI}_{after})	$P^{D}_{improved}$ ($P^{CI}_{improved}$)
/users/search	inefficient iteration	31	0.36ms (0.19ms)	0.26ms (0.13ms)	28.8% (27.79%)
/users/search/id	inefficient iteration	31	1.7ns (2.5ns)	1.19ns (1.63ns)	29.53% (34.8%)
/api/ladywithpet	misused APIs	18	5.89ms (2.74ms)	4.99ms (2.24ms)	18.03% (18.13%)
/api/thedea	misused APIs	18	5.63ms (2.71ms)	4.82ms (2.25ms)	14.39% (15.3%)
/api/theredroom	misused APIs	18	0.65ms (1.87ms)	0.53ms (1.56ms)	18.06% (16.58%)
/api/thegift	misused APIs	18	1.17ms (0.36ms)	1.04ms (0.31ms)	11.03% (12.71%)
/search_leak	memory leak	24	619.10kb (476.16kb)	519.13kb (409.10kb)	16.15% (14.17%)
/properties/favorites	memory leak	42	824.62kb (669.4kb)	511.37kb (201.7kb)	61.26% (69.87%)

on the specific choice of running environments. Also, the ULOC for the patches automatically generated by CANDOR can differ in size from those generated by humans. Because CANDOR generates patches at statement granularity, no additional lines can be added for readability or commenting. Human programmers are free to format the patches in an arbitrary fashion, thus affecting the total number of lines taken by their bug fixing patches.

4.4 Conclusions and Future Work

We have presented a new debugging approach—CANDOR—that facilitates the debugging of full-stack JavaScript applications. As a future work direction, we plan to conduct a systematic user study of JavaScript programmers to assess the effectiveness and usability of the CANDOR debugging approach.

Chapter 5

Correcting Distribution Granularity

5.1 Assessing and Improving the Utility of Distributed Functionality

We target distributed applications that comprise the client and server parts, communicating with each other by means of distribution middleware, such as the HTTPClient library or CORBA [95]. Our application domain are *full-stack JavaScript applications*, in which both the client and server parts are written in JavaScript; this domain is becoming increasingly widespread due to the popularity of Node.js and other server-side JavaScript frameworks. The client invokes server-side remote functionality, which executes corresponding code and returns back the results to the client. The client passes input parameters, and the server returns results. The middleware mechanism serializes and deserializes both the parameters and results to transfer them across the network and make them available for computation.

5.1.1 Motivating Example

Consider *Bookworm*, a book reader implemented as a full-stack JavaScript mobile app. In addition to enabling users to read books on their mobile devices, *Bookworm* has a feature that reports statistical information extracted from the text of the books. To that end, the app features a remote service that given a book title, analyzes its text and returns the

results of this analysis. Because text processing is computationally intensive, it is commonly performed remotely at a powerful server rather than locally on a mobile device. The original implementation of this remote text analysis service runs all analysis tasks (e.g., overall length, punctuation percentage, unique vocabulary, etc.) in sequence, returning their results in bulk. For large books, waiting for all the tasks to complete before any results become available can degrade the user experience. Hence, a possible restructuring could separate the unit containing all sequentially performed analysis tasks into multiple asynchronous units, each of which immediately returning the computed results back to the client.

As it turns out, the majority of the resulting remote executions for analysis tasks (e.g., overall length, punctuation percentage, etc.) are not computationally intensive. However, the client consumes additional resources to execute these tasks by performing multiple remote invocations. The only analysis task that involves heavy processing and takes a long time to execute is "extract unique vocabulary." To minimize the overall latency of invoking these text analysis tasks, they can be restructured into two remote services: one to invoke "extract unique vocabulary" and the other one to invoke the remaining analysis tasks in bulk (See Figure 8.1).

To determine the optimal structuring and distribution of the text analysis tasks would require profiling their execution under different inputs. Hence, the remote analysis services need to be both restructured and redistributed, a non-trivial re-engineering task. The approach presented herein systematically identifies what an optimal distribution is for a given optimization criteria and presents automated program transformations that eliminate much of the engineering complexity of redistribution.

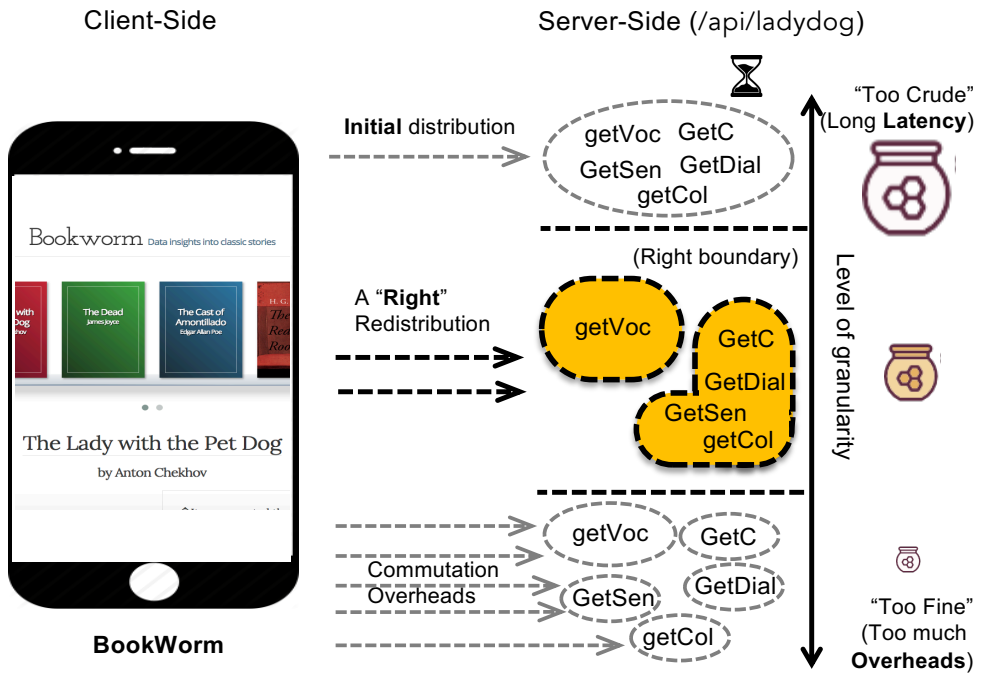


Figure 5.1: Correcting Granularity of *Bookworm*

5.1.2 Distribution Execution Cost Function

In the motivating example above, we show how redistribution can optimize remote services. Rather than trying to determine redistribution optimization opportunities through trial-and-error, distributed app developers need intuitive numerical models that can inform them about the actual cost/benefit ratio of remote services. To that end, we formulate the *distribution execution cost function*.

Problem Formulation: Our goal is to determine which functional distribution from the client's standpoint would minimize the following cost of distributed execution function:

$$C^{Dist\_Exec}(r_i) = \alpha \cdot latency(r_i) + (1 - \alpha) \cdot \sum resource(r_i).$$

Intuitively, executing a functionality remotely reduces the computational load on the client

at the cost of the delay, measured as the remote invocation's latency, and the local resources consumed to make this invocation. Typically it is distribution middleware that consumes these additional resources, including computation, memory, and energy. The invocation latency measures the expected deterioration in the user experience—the more time the user has to wait for a remote functionality to complete, the less satisfying their experience will be. Hence, the distribution cost is the sum of the expected deterioration in the user experience and the amount of additional local resources consumed to invoke the remote functionality. Hence, the optimization objective is to identify the most favorable remote utility/client cost ratio. Our optimization strategy strives to determine the level of granularity for remote services that maximizes this ratio.

Consider a long-running bulk remote service r. Since invoking this service takes a long time, even with low client-side resource utilization, the client cost may not be as favorable. One can break up this long-running service r into a collection of smaller services $r_1^{'}, \ldots, r_k^{'}$. Assume that these smaller services are not inter-dependent and now can be invoked asynchronously. First of all, the original computational work being offloaded to the server or the *utility* for r remains unchanged by this redistribution. However, the combined latency of invoking these smaller services would decrease, but the consumed client resources would increase due to multiple invocations, so the resulting cost (or utility/client cost ratio) may not decrease. A more optimal redistribution in this scenario may be to combine some of these smaller remote services into one to decrease the resources that the client would consume to invoke them. Hence, the cost of distribution function is defined as the sum of the normalized execution latency and client-consumed resources required to invoke a remote service. It is the weight factor α that normalizes the latency and resource consumption terms.

Problem Solution Outline: We estimate an optimal distribution for a remote service by minimizing the cost of invoking the service's constituent functionalities. To that end,

these functionalities can be invoked individually or in bulk in different combinations. The following two operations express the required program restructurings:

- $[r'_1, \ldots, r'_k]$=*partition*(r): partitions a remote service r into k independent parts, each of which becomes an individually invocable remote functionality.
- r_h=*batch*$([r_0, \ldots, r_n])$: batches n remote functionalities into a larger remote service r_h.

Notice that the *batch* operation may be applied multiple times to different remote services to achieve the required service combinations. D-Goldilocks implement a divide & conquer algorithm that by means of *partition* and *batch* identifies a distribution that minimizes the C^{Dist_Exec} function.

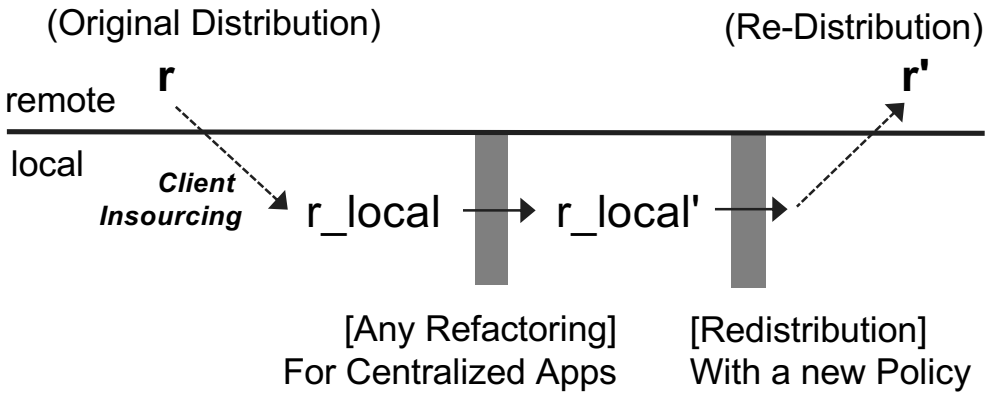


Figure 5.2: Step I: Generating a centralized Variant of a remote service

5.1.3 Client Insourcing to Restructure Remote Services

The *partition* and *batch* operations work with regular JavaScript functions rather than remote services. Hence, to transform remote services into local functions, we introduce a domain-specific automated refactoring—*Client Insourcing*, which given a remote functionality invoked via middleware, integrates that functionality with the client code. Client

Insourcing undoes the current distribution r, thus creating a centralized variant r_local that can be redistributed differently into $r^{'}$ by means of a refactored local version $r\_local^{'}$ (Figure 5.2).

Section 5.2.1 describes the technical details of Client Insourcing.

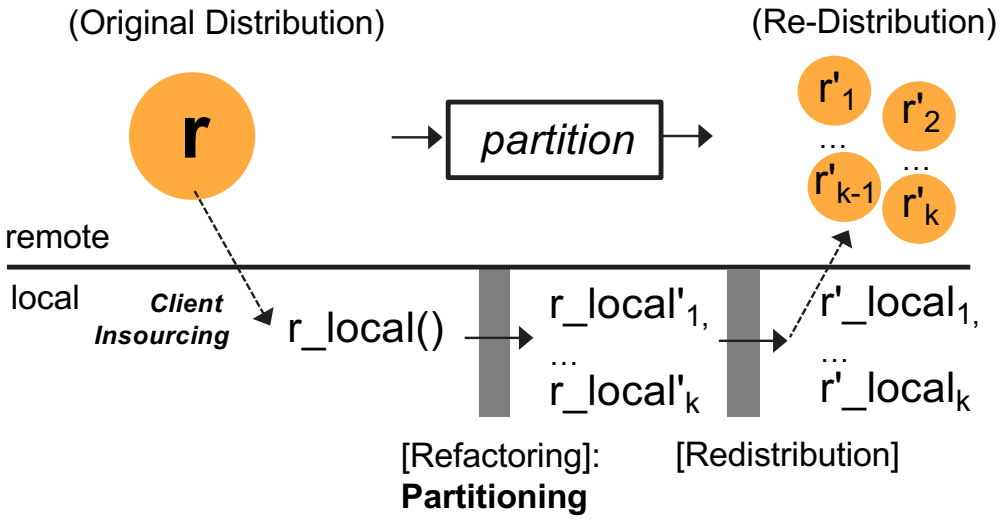


Figure 5.3: Step II: Partitioning the variant into independent units

5.1.4 Partitioning Insourced Functions

Based on the ability of Client Insourcing for restructuring, our approach can partition a remote service into multiple remote parts (Figure 5.3). Assume a remote function r consists of some distinct functionalities. To identify independently invocable parts, we can perform Client Insourcing on r, making it a centralized function r_local. Then, we can apply program analysis and refactoring for centralized apps to split the r_local into k distinct functions $r\_local^{'}_1, \cdots, r'\_local^{'}_k$. Finally, k distinct functions can be distributed becoming $r^{'}_1, \cdots, r^{'}_k$.

5.1.5 Batching Remote Invocations (BRI)

For networking environments with large bandwidth and high latency, it may be advantageous to batch multiple remote invocations into a single one to reduce the aggregate latency. The research literature describes several approaches that implement this optimization. The Data Transfer Object (DTO) and Remote Façade [33] design patterns aggregate individual services. Remote Façade exposes multiple fine-grained services via a coarse-grained remote interface. DTO serves as a bulk object for transferring parameters and results of remote service invocations. Remote Batch Invocations (RBI) provides language support for creating DTOs for combining the invocation of arbitrary remote services, which can also be intermixed with local operations [45].

Our approach batches fine-grained distributions by automatically generating Remote Façades. First, the small remote functionality is insourced to become local. Then, the resulting independent local functions are inlined into a single function using the *Inline Function* refactoring. That single function then becomes a new unit of distribution.

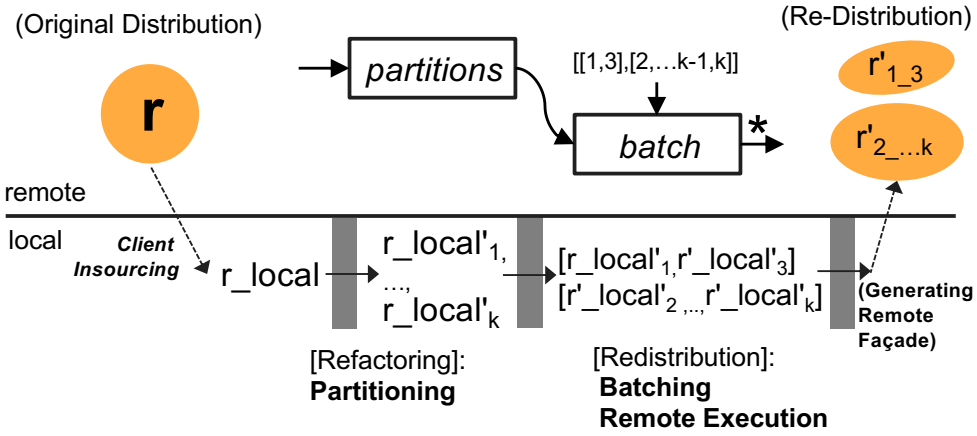


Figure 5.4: Batching remote services

Combining the *partition* operations, our approach can arbitrarily generate Remote Façades for k distinct functions $r'\_local_1, \cdots, r'\_local_k$. For instance, a subset $\{r'\_local_1, r'\_local_3\}$

are inlined into a single function $r'\_local_{1\_3}$, which is then distributed into r'_{1_3} (Figure 5.4). In essence, a combination of these automated refactorings creates a distributed execution that would be similar to the result of implementing the Remote Façade optimization by hand.

5.2 Reference Implementation: D-GOLDILOCKS

We concretely realized our approach as a series of automated refactorings. These refactorings both migrate functionalities between different hosts and also restructure the granularity of remote service invocations.

5.2.1 Client Insourcing Refactoring

Client Insourcing is a domain-specific[1] refactoring that automatically integrates a remote functionality with the client code. Because our application domain is JavaScript applications, and JavaScript is known to defeat static analysis techniques, Client Insourcing includes a dynamic analysis phase to identify the exact boundaries of the server functionality to insource. The programmer is only required to annotate the invocation points of the remote functionality to insource. These points correspond to the locations in the client code, at which remote invocation parameters are serialized to be transferred across the network, and the remote service's results are unserialized to be used in the subsequent program steps. Intuitively, Client Insourcing identifies serialization/unserialization points in the client code is to detect the entry/exit execution points of the remote functionality to insource. To that end, Client Insourcing extracts the passed parameters and the return results in the recorded HTTP traffics.

[1] its application domain are full-stack JavaScript applications

Algorithm 1 Generating Centralized Variant

Client_Insourcing (C_{src}, S_{src}) **Input** : C_{src}:client code, S_{src}:server code
Output: $C_{src}^{insourced}$: Centralized App for C_{src}
S_{n_src}=normalize(S_{src}) `/* add Instrumenting code(Jalangi2/JS code)` */
[C_{src}^{inst},$S_{n\_src}^{inst}$]=addInstrument(C_{src},S_{n_src}) `/* Instrumenting the entry and exit points` */
Log=remoteExecution(C_{src}^{inst}, $C_{n\_src}^{inst}$) `/* Loading JavaScript program rules` */
$Rule_{nodejs}$=loadNodejsProgramRules() `/* Generate facts for server code` */
$Fact_{S_{src}}$=genProgramFact(S_{n_src})
$Model_{S_{src}} = Rule_{nodejs} + Fact_{S_{n\_src}}$ `/* Check dependency for candidate points` */
`/* Query dep. stmts for entry/exit` */
$Stmts_{entry}$=queryDepStmt($Log.P_{entry}$,$Model_{S_{src}}$)
$Stmts_{exit}$=queryDepStmt($Log.P_{exit}$,$Model_{S_{src}}$) `/* Cutting dependent JS statements` */
$Stmts_{dep}$=$Stmts_{exit} - Stmts_{entry}$ `/* Make a regular ftn with adaptation` */
f_{local}=compGen($Stmts_{dep}$, $Log.(P_{entry}, P_{exit})$) `/* Add the` f_{local} `with Logged Position` */
$C_{src}^{insourced}$=compAdder(f_{local},$Log.pos$,C_{src}) `/* Return insourced version of` C_{src} */
return $C_{src}^{insourced}$

As introduced in Chapter 3, the design *Client Insourcing* of follows that of other declarative program analysis frameworks [57, 62, 98] that analyze JavaScript using the z3 SMT solver. To analyze server code written by means of the Node.js framework, Client Insourcing defines its own sets of z3 rules for Node.js and that of facts for subject programs. The profiled parameters and return results are added as new z3 facts to be able to reason about the entry and exit points of the remote execution. Client Insourcing generates local functions by solving constraint problems with z3. First, Client Insourcing checks the dependency between the entry and exit point candidates to locate the correct pair of points. Next, it finds a subset of dependent statements for the exit point in the server part and the subset of the dependent statements from the entry point. Client Insourcing generates local functions by differencing the entry and exit sets. Finally, the generated functions change the call structure of the client code into regular local calls by using the recorded insertion points. Algorithm **??** shows the overall refactoring procedure.

```
//original Client:app.js
$scope.getLadyWithPetDog = function()
    {...
  $http.get('/api/ladypet')
  .then(function(resp){
     var text = resp.data; ...
    });/*remote invocation*/}
//original Server:server.js
function getSenAvg(array){...};
function getVoca(str){...};
...
app.get('/api/ladypet',
   function(req, res){...});
```

```
//after Client Insourcing:app.js
//Insourced remote functions
function getSenAvg(array){...};
function getVoca(str){...};
...
function ladypet_local(){
//invoke every subtasks
...};
$scope.getLadyWithPetDog = function()
    {...
    //from remote to local
    var text = ladypet_local();
...}
```

Figure 5.5: Redistribution Step I: Client Insourcing of Bookworm

5.2.2 Partitioning a Function into Individually Invoked Functions

To partition a JavaScript function into individually invoked functions, D-GOLDILOCKS first applies static analysis to determine the dependencies of the function-to-partition. These dependencies comprise all references to global references and the invocations of other functions. To that end, D-GOLDILOCKS traverses the function's control-flow graph[2] in the depth-first order. Then a greedy algorithm is applied to determine the maximum number of partitions, each of which is an independently invocable function. The algorithm strives to produce the highest number of candidate partitions, with the following exceptions: 1) mutually dependent partitions as indicated by the original function's call graph or 2) partitions that share global variables. Such candidate partitions are merged into a single one.

[2]https://github.com/wala/JS_WALA

5.2.3 Batching Remote Invocations

D-Goldilocks automatically generates a client-side DTO and remote Façade stubs for batching the small remote services. The actual Remote Façade function, invoking the original services, becomes the new entry point of the remote execution. The client DTO stub accumulates the remote invocations of the fine-grained services at the client before transferring them in bulk to the remote Façade function; the `BATCH_PARAMETER` parameter to the batch specification becomes the number of service invocations to accumulate. The remote Façade function sequentially (or synchronously) invokes the bundled services and returns their execution results combined into a single value in bulk. For the following specification, D-Goldilocks generates a remote Façade `f1name_f2name` with the concatenated function names of the original fine-grained services `f1name` and `f2name` (Figure 5.6).

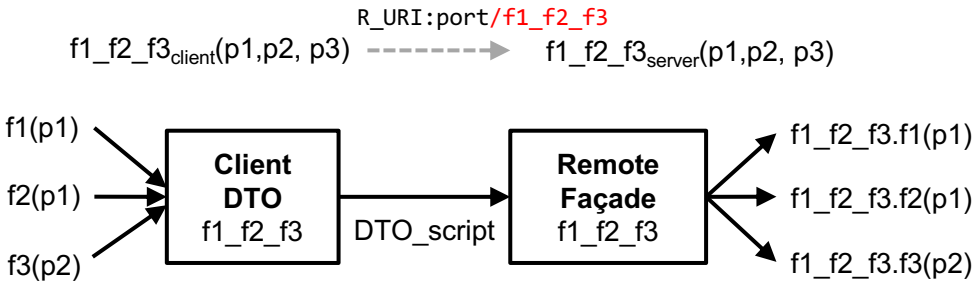


Figure 5.6: Batching Invocation of sub-functions with the Client DTO and Remote Façade

```
//after Redistribution: www/index.html
<!DOCTYPE html>
<script src="./app.js"> ...
 ClientDTO.batch_param = BATCH_PARAMEMTER;
//Batched Invocations:
 getSentenceAvg = ClientDTO(getSentenceAvg);
 getVocabularies = ClientDTO(getVocabularies); ...
</script>
```

Figure 5.7: Redistribution Step II: Batching Insourced Functionality

5.2.4 Redistribution Steps

The three code snippets in Figure 5.5 show the original client and server parts, their centralized insourced version, and a redistributed client part.

Client Insourcing automatically transforms a client server distributed interaction into a centralized counterpart, moving server functions to the client and replacing all middleware invocations with local function calls. When insourcing a remote functionality, all its dependent server-side code has to be copied to the client. That code may be scattered around multiple functions and standalone declarations. Each insourced remote functionality is placed in a single function, added to the client codebase. Client Insourcing is similar to the *Extract Function* refactoring. Both refactorings create a newly named function and the call sites to invoke it. Client Insourcing differs in moving the extracted code from the server to the client and replacing middleware functionality with local calls. The middle column of Figure 5.5 shows the centralized variant produced by Client Insourcing the code in the left column.

This centralized variant is used for profiling and redistribution. In this example, two of the original server functions are batched into a single function, invoked in the same remote roundtrip. The batching operation is implemented via the *Data Transfer Object* (DTO) pattern on the client and the Façade pattern on the server.

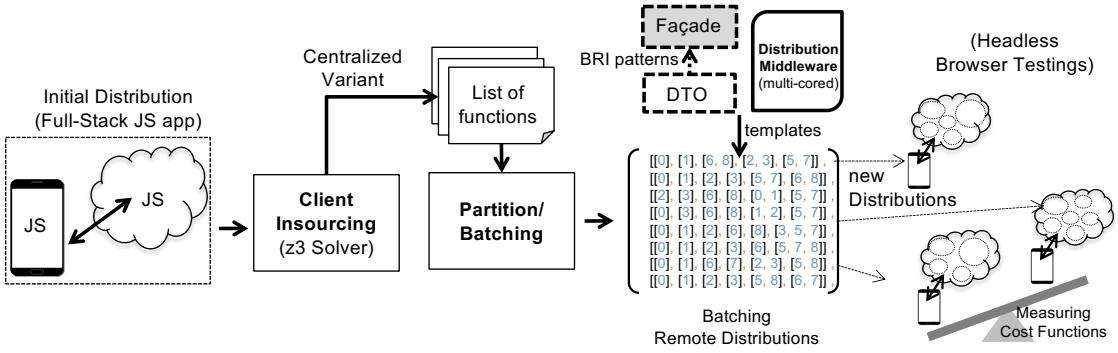


Figure 5.8: Process for D-GOLDILOCKS

5.2.5 Distribution Framework: Transforming Local Functions into Remote Services

D-Goldilocks implements a framework for seamlessly transforming local JavaScript functions into remote services. D-Goldilocks maintains a list of local (insourced) functions that implement the business requirements. D-Goldilocks uses the mustache.js framework[3] to generate different client/server combinations of the insourced functions. The resulting client and server parts communicate with each other by means of Ajax and the Express.js middleware, as the majority of our subject apps already use this middleware. For the server to explicitly handle concurrent executions, it is enhanced with a multi-core engine[4] for the node.js. These frameworks introduce the required distribution with minimal changes. The newly redistributed functions only need to unmarshal their parameters and marshal their results.

Figure 5.8 shows our overall automated refactoring approach.

5.3 Evaluation

Our evaluation seeks answers to the following questions:

- **RQ1:—Value:** How much programmer effort is saved by D-Goldilocks's automatic redistribution operations?
- **RQ2:—Cost Model Correctness:** How applying the partition and batch operations affect the distributed execution's "latency" and "consumed resources" attributes?

[3]https://github.com/janl/mustache.js
[4]http://learnboost.github.io/cluster

- **RQ3:—Utility of Cost Model for Redistribution:** How useful is the cost function for guiding redistribution decisions?

- **RQ4:—Energy Consumption:** What is the effect of redistribution on the amount of energy consumed by the client?

5.3.1 Evaluation Setup

Dataset

Our evaluation subjects are real-world full stack distributed mobile JavaScript applications and benchmarks from the *extremeJS* [106]. *extremeJS* built remote services over their distributing framework focusing on JavaScript offloading. We tested their remote functionalities only changing the server middleware from their distributing framework (V8 within a C++ app) into Express.js.

Latency and CPU Utilization of Remote Services

We profile remote services under standard loads in terms of the latency and resources for the client. We use a V8 profiler[5], which supports the line-by-line performance profiling of JavaScript programs. D-GOLDILOCKS injects probes into the instrumented source code and collects samples, which contain the execution times (L in Table 5.1) and CPU utilization levels for each block. By summing these CPU levels, We calculate the resource consumed by a remote execution ($\sum T_{cpu}$ in Table 5.1). Computationally intensive benchmarks with remote functionalities always exhibit a high latency. We ran our measurements over headless

[5]https://github.com/node-inspector/v8-profiler

browser testing frameworks[6][7] to emulate a real world's web client applications. The remote server is hosted by Dell-OptiPlex5050 and the client execute remote services within the same machine.

Table 5.1: Subject Remote Services for Evaluating D-Goldilocks

Remote Ser	L(ms)	$\sum T_{cpu}$	f_{CI}^{LOC}	$f_{sub}^{ind}(f^{decl})$	$\|D\|$
/api/ladypet	77.38	337	394	8(9)	1.6M
/api/thedea	164.62	695	394	8(9)	1.6M
/api/thered	42.96	370	394	8(9)	1.6M
/api/thegift	37.69	390	394	8(9)	1.6M
/api/bigtrip	42.11	304	394	8(9)	1.6M
/api/offshore	30.82	400	394	8(9)	1.6M
/api/wallppr	56.2	396	394	8(9)	1.6M
/api/thecask	20.6	432	394	8(9)	1.6M
/string-fsta	29.85	328	38	2(5)	76
/cflow-rec	35.43	326	49	3(4)	245
/pprty/brkrs	20.64	323	379	3(3)	1.5K
/pprty/brkrId	15.62	332	382	3(3)	1.5K

5.3.2 Evaluating Software Engineering Value

Programmer Effort Saved

To answer **RQ1**, we estimate the value of D-Goldilocks automatically generating JavaScript code. As an automated refactoring, Client Insourcing saves programmer effort required to move remote functionality to the client, so it can be invoked via local function calls. We count the number of uncommented lines of JavaScript code (LOC) that need to be edited by hand to perform the refactoring. Notice that Client Insourcing transformations involve two phases: generating local functions and replacing middleware invocations with local calls. The local functions are generated by copying the server-side code, which becomes the body of new

[6]https://github.com/GoogleChrome/puppeteer
[7]https://github.com/jsdom/jsdom

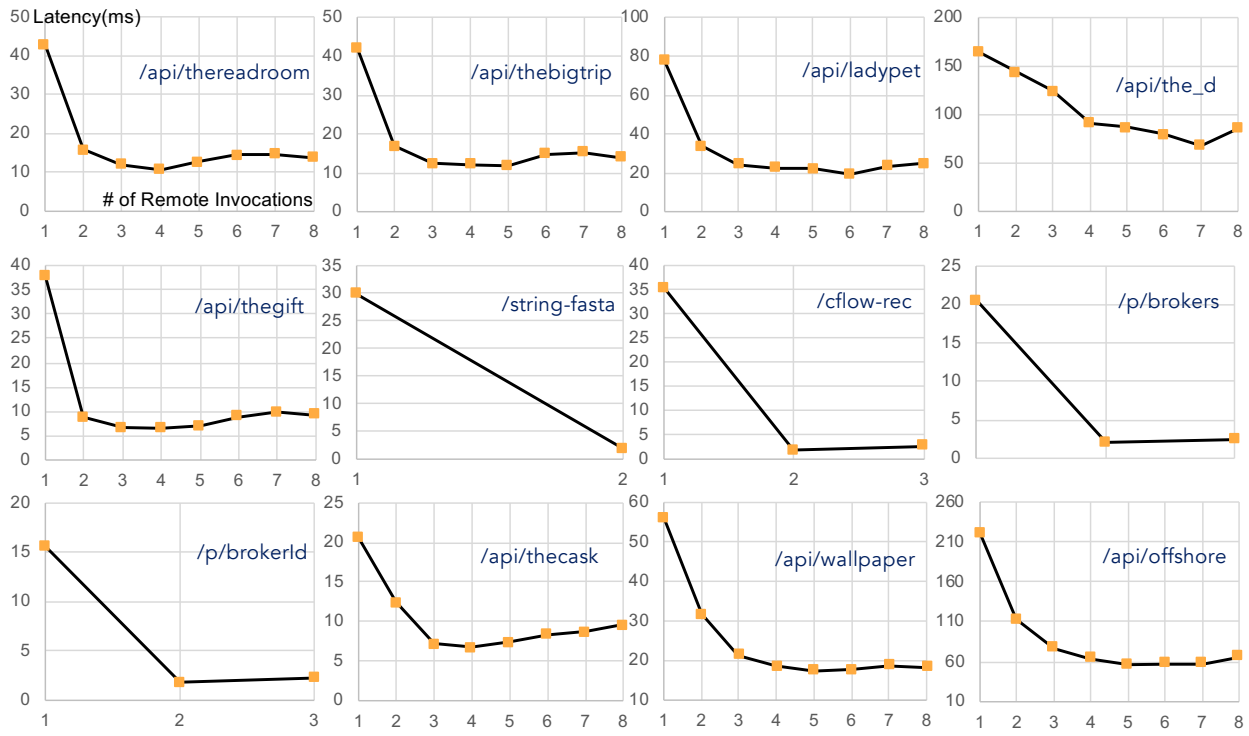


Figure 5.9: Latency(ms) versus the number of Remote Invocations

client-side functions, whose parameters and returns values are automatically inferred from the corresponding server entry/exit points (f_{CI}^{LOC}). The original middleware functionality is replaced with local calls. For instance, Client Insourcing the `/api/ladypet` remote service generates a local function of 394 ULOCs. f^{decl} is the number of function declaration for f_{CI}^{LOC}, originally all these sub functions are grouped together in the remote execution. Another D-GOLDILOCKS's operation is partitioning an insourced function f^{LOC_CI} into smaller individually invoked functions $f_{CI\_1}^{LOC}, ..., f_{CI\_n}^{LOC}$. For each subject, we report the number of the resulting functions f_{sub}^{ind}, where $f_{sub}^{ind} \leq f^{decl}$. The final D-GOLDILOCKS's operation is batching individually invoked functions into a larger function. To be able to determine what the optimal combination of function is, D-GOLDILOCKS generates all possible combinations of individually invoked functions. Hence,we estimate the saved manual programming effort, $|D|$, as the product of f_{CI}^{LOC} and all possible combinations of f_{sub}^{ind}. Because of the combinatorial

explosion, the values of $|D|$ tend to be too large for any reasonable manual treatment. For example, $|D|$ for `/api/ladypet` is $394 \times 4,139 \cong 1.6 \times 10^6$ ULOCs.

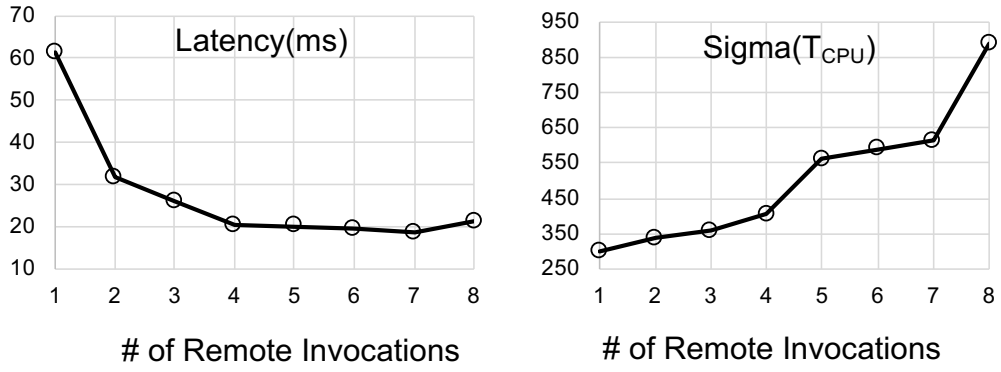


Figure 5.10: Scales between Latency and CPU Usage

Refactoring Impact

D-Goldilocks redistributes a remote functionality by insourcing it, partitioning it into parts, and batching these parts into new individually invoked remote functionalities. In this experiment, we assess the actual performance impact of the number of batched parts on the resulting invocation latency and consumed resource (**RQ2**). Figures 5.9 and 5.10 show the observed metrics for our experimental subject applications. The larger the number of new remote functionalities, the smaller is the aggregate average latency incurred by invoking them. The latency drops precipitously as the number of functionalities start growing, but than flattens due to the additional overhead of multiple remote invocations. Whereas, the overhead or the CPU usage proportionally increases with the number of new remote functionalities.

Utility of Redistribution Cost Model

To answer **RQ3**, we applied our cost function to different redistribution scenarios of our subjects. We empirically determined the required normalizing factor for the latency(milliseconds)

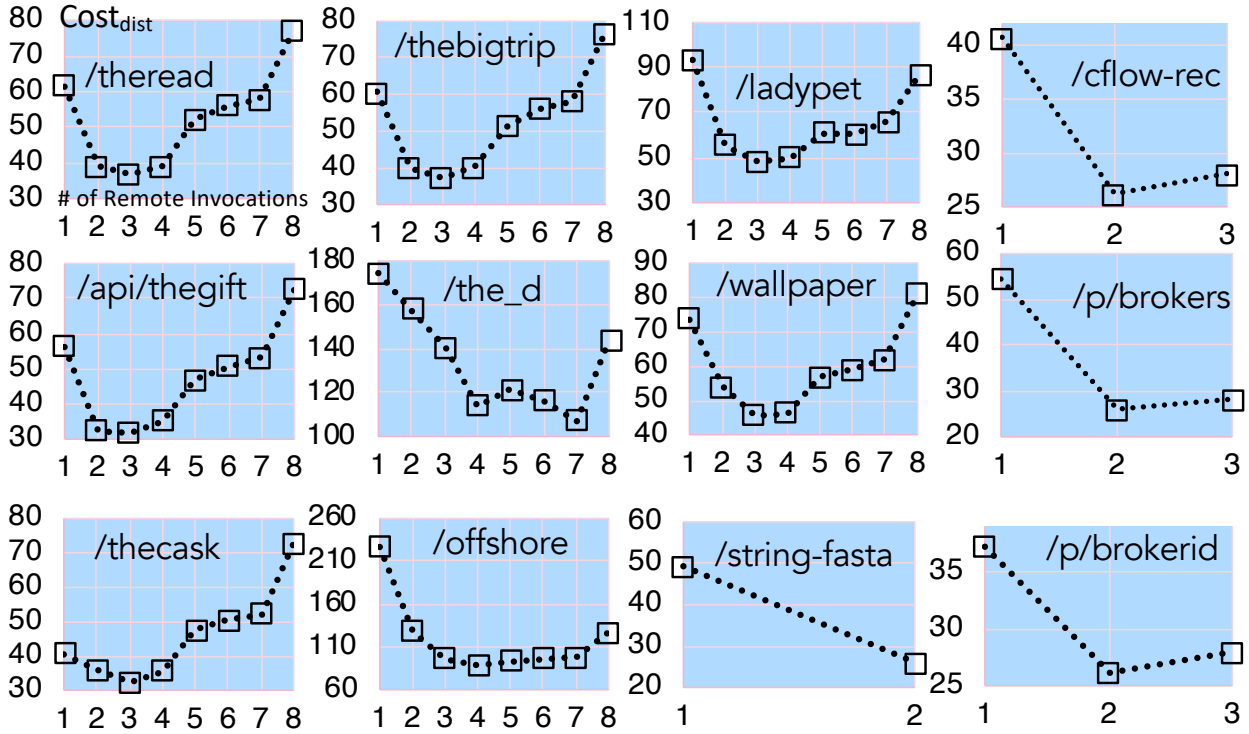


Figure 5.11: Cost Functions versus the number of Remote Invocations

and sum of CPU usages terms by scaling the observed latency/CPU usage ratios across all measurements (See Figure 5.10).

$$C^{Dist\_Exec}(r_i) = \alpha \cdot L(r_i) + (1 - \alpha) \cdot \sum T_{cpu}(r_i)$$

, where $\alpha = \overline{L}/\overline{\Sigma T_{cpu}} = \mathbf{0.9281}$.

Splitting a singe long-running remote function into a small number of asynchronously invoked parts decreases both the aggregate latency and cost. However, as the number of partitions grows, so does the cost, due to the increasing overhead of invoking multiple remote functions (Figure 5.11).

Figure 5.12 shows how two the optimal distributions of `/api/theread` and `/api/thegift` bring the distributed execution cost down to the minimums. Recall that the task of getting unique

vocabulary (`getVocabulary`) was relatively computationally intensive, as compared to other tasks.

The optimal distribution comprises three individually invoked remote services, extracted by partitioning `getVocabulary` into smallest possible functions and then batching them to minimize the aggregate latency and CPU utilization.

5.3.3 Evaluating Performance and Energy Consumption

To answer **RQ4**, we measure the amount of energy consumed by a mobile device to execute remote services over a stable WiFi network. The client device, QISKIW-L24-Huawei, runs Android Marshmallow, and the remote server is hosted by Dell-OptiPlex5050. We use PowerTutor [112], a model-based energy profiler for mobile apps, to estimate energy consumption (EC).

We report on the energy amounts consumed in three deployments of the *Bookworm* application: (1) the original distributed execution, $EC_{original\_dist}$, (2) the best distributed execution achieved via redistribution, EC_{best_dist}, and (3) the worst distributed execution achieved via redistribution, EC_{worst_dist}. Figure 5.12 shows the best distribution (2 total remote invocations). The worst distribution makes 8 remote invocations, while the original version makes 1 remote invocation. The ✸ symbol designates this subject's heavy processing function, whose execution time dominates the total execution. As it turns out, the original distribution consumes the lowest amount of energy, $EC_{original\_dist}$=8.4mJ, with the best distribution not far behind, EC_{best_dist}=13.4mJ. The worst distribution is an energy guzzler, consuming 6 times as much energy as the original version, EC_{worst_dist}=47.4mJ.

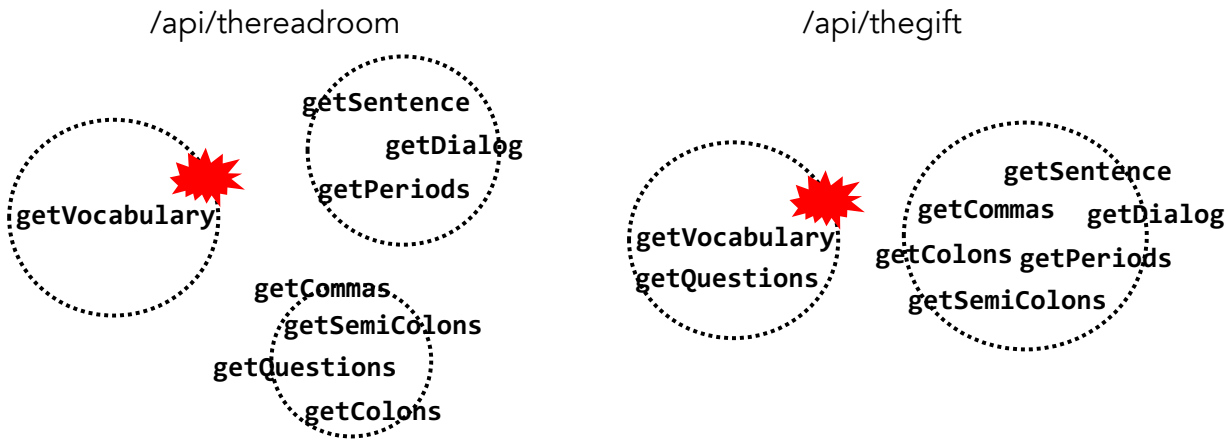


Figure 5.12: Examples of Optimal Distributions

5.4 Discussion

Our experimental results are subject to both internal and external threats to validity. Our approach also has applicability constraints. We discuss these and other issues in turn next.

5.4.1 Internal Validity

To redistribute our subject applications, we use the Express, JQuery, and Cluster frameworks. The way these frameworks introduce distribution may certainly affect the performance of the resulting distributed applications. By using these frameworks in a black-box fashion, we have no control over how they implement their remote execution features. Since our redistribution phase starts from a centralized JavaScript variant, any other distribution frameworks can be used in place instead, possibly resulting in differently performing distributed applications. Nevertheless, these differences would be unlikely to change the overall performance profile of the redistributed applications. As our measurements show, the performance latency of remote invocations is dominated by network communication and the server's computational load. The choice of distribution middleware would have a marginal

impact on the performance of these functionalities.

As our units of distribution, we use existing functions. Another possibility would be to consider splitting existing functions into smaller units that can be distributed independently. Measuring the performance of and redistributing code at the level of granularity of existing functions certainly have impacted our performance results. Nevertheless, in this work we aim at a fully automatic approach to determining which distribution would be optimal. It would be impossible to automate the process of breaking up existing functions into meaningful constituent blocks.

5.4.2 External Validity

D-Goldilocks makes all redistribution decisions based on the obtained performance characteristics of the application, whose remote functionality has been insourced. Our implementation relies on the V8 profiler to measure the performance of such applications. It is possible that other profilers could show different performance numbers, thus affecting D-Goldilocks's redistribution recommendations. Network connectivity can also affect our experimental results. All our experiments were conduced over a stable WiFi network connection. Operating over limited unstable networks would incur higher energy consumption overheads. D-Goldilocks applies a cost function to decide whether a given distribution needs to be fine-tuned. One may disagree with this heuristic and choose a different one, particularly well-suited for certain application domains. Even if one completely rejects the validity of our decision-making heuristic, our overall redistribution approach still has value. The ability to reshape centralized functionality before redistributing the result is a new promising approach to optimize the execution of distributed applications.

5.4.3 Applicability and Limitations

Distributed execution is always a result of certain architectural decisions. D-GOLDILOCKS makes it possible for developers to revisit these decisions, without resorting to prohibitively expensive manual code modifications. Instead, D-GOLDILOCKS relies on domain-specific and general refactoring transformations. Hence, developers who use D-GOLDILOCKS are still required to understand the original distributed application's architecture. As is usually the case, D-GOLDILOCKS eliminates much of the *accidental* rather than *essential* complexity of architecting distributed applications [17].

All our evaluation examples are full-stack JavaScript applications. However, conceptually our approach is quite general and should be applicable to any distributed application domain. However, other domains may require additional engineering effort. Although full-stack JavaScript applications have become extremely popular, to redistribute JavaScript code to execution platforms that use a different programming language, one may be able to apply language-to-language translation, an approach whose success can differ widely depending on the source and target languages.

Chapter 6

Improving the Responsiveness of Web apps

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 [101]. 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 [104].

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 [11, 51, 94, 106]. 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 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:

6.1 Approach

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

6.1.1 Motivating Example

Consider *Bookworm*, 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.

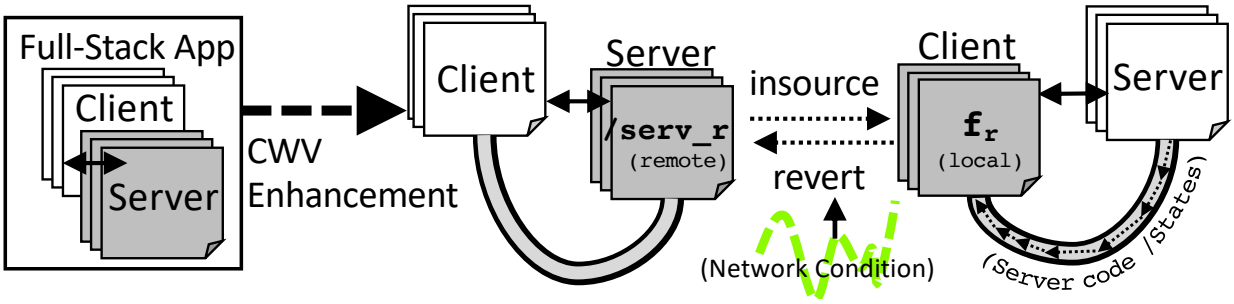


Figure 6.1: Conceptual View of Communicating Web Vessels (CWV)

6.1.2 Approach: Communicating Web Vessels

To optimize the performance of mobile web apps for the current network conditions, **CWV** continuously applies the two operations depicted in Fig. 6.1:

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.

2. $revert(f_r)$: The client stops locally invoking the insourced partition f_r, and starts remotely invoking its original server version $/service\_r$.

6.1.3 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 6.2.3.

$$RT(f_r) = \begin{cases} T_{server}(f_r) + RTT^{net} & \text{remote exec.,} \\ T_{client}(f_r) & \text{local exec.} \end{cases} \tag{6.1}$$

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

6.2 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 6.2.1 and 6.2.2). Furthermore, our

approach establishes an efficient protocol for the transformed app to switch between different execution modes (Section 6.2.3), transferring the relevant code correctly and safely (Sections 6.2.5 and 6.2.6).

6.2.1 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. 6.2-(a)). To that end, **CWV** uses Jalangi [88], 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

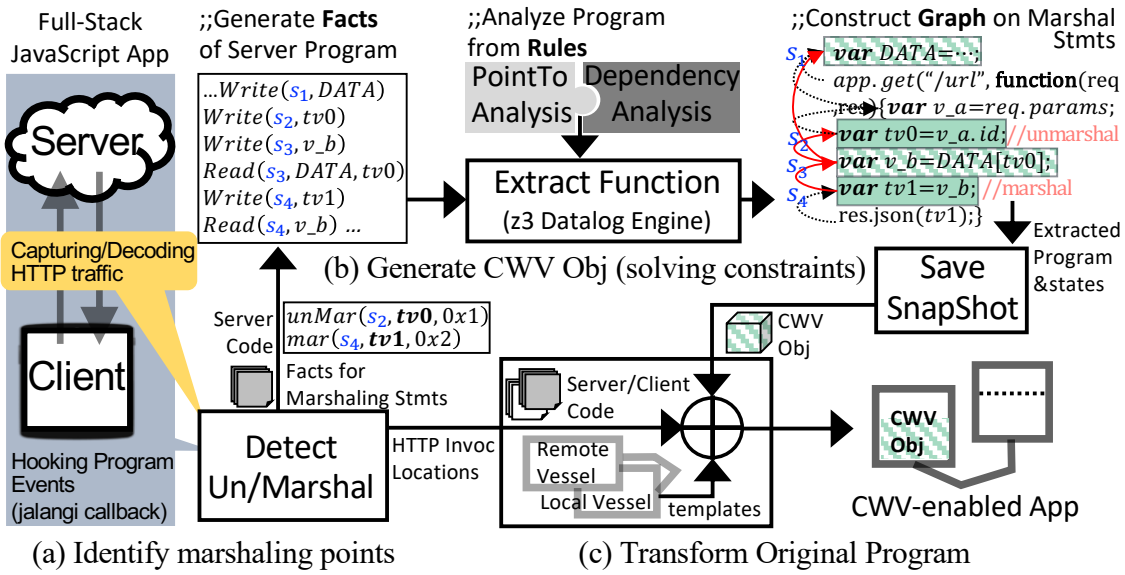


Figure 6.2: Automated Program Transformation for enabling CWV

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 [5, 98]. 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 [23] and then extracts them into a CWV-specific object that is movable between vessels (Fig. 6.2-(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 `mysql_server.query(SQL_STATES)`, 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_STATES)`. 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. 6.2-(c)).

6.2.2 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. 6.3). 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*. This object is used for storing functions and other JavaScript objects received from the server[1]. 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 (6.1). 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

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

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,

$$rcwv.main, rcwv.ftns[0], \cdots, rcwv.ftns[k] \tag{6.2}$$

and their corresponding saved states for global variables

$$rcwv.gvars[0], \cdots, rcwv.gvars[l]. \tag{6.3}$$

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

$$rcwv.sql[0], \cdots, rcwv.sql[m]. \tag{6.4}$$

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.

6.2.3 Updating Modes and Cutoff Latency

The transition diagram in Fig. 6.3 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 6.2.2). 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 [47], CWV filters out such temporary fluctuations by applying an adaptive filter [69], which calculates the covariance matrices and noise values for RTT^{net}_{raw} and then estimates the RTT^{net} metric in Equation (6.1).

In particular, CWV uses an adaptive filter, which repeatedly calculates the covariance matrices for the RTT and noise values. To provide the adaptive filter for CWV, we created a WebAssembly module (wasm) that encapsulates a third-party implementation of this filter in

the Go language. This wasm module exposes function `estimateRTT`, which is continuously invoked by the mobile browser, with the wasm mechanism ensuring a low invocation overhead. See Figure 6.3 for the logic that determines how frequently this function is invoked.

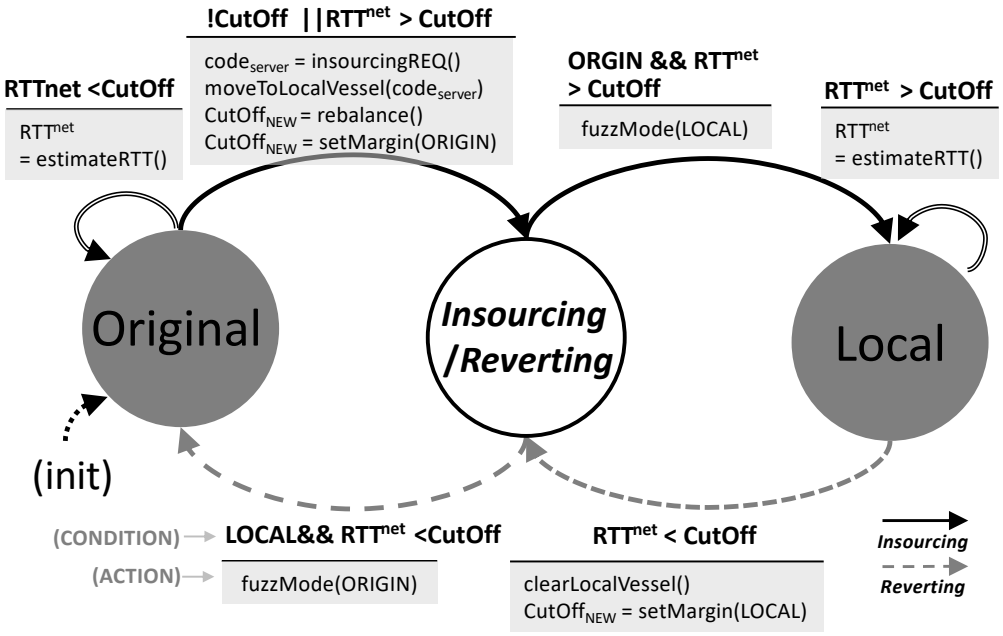


Figure 6.3: Transition Diagram for CWV-enabled Client

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^{NET}_{cutoff}. 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. The algorithm in Algorithm 2 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.

Algorithm 2 Updating Switch Point and Transitioning Mode

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 $m^{(k+1)}$

//Remove spike by adaptive Kalman Filter

$RTT^{net}_{filtered} \leftarrow estimateRTT(RTT^{net}_{raw})$

if $RTT^{net}_{filtered} > \tau^{NET(k)}_{cutoff} \&\& m^{(k)} == Origin$ **then**

  //Profiling the difference for $\overline{T} : rebalance()$

  $\tau^{NET(k+1)}_{cutoff} \leftarrow \overline{T}_{server}(f_r) - \overline{T}_{client}(f_r)$

  //Set margin to the next cutoff condition

  $margin \leftarrow (1-\theta) \cdot RTT^{net}_{filtered}$;

  $\tau^{NET(k+1)}_{cutoff} \leftarrow min(\tau^{NET(k+1)}_{cutoff}, margin)$

  $m^{(k+1)} \leftarrow Local$

end

if $RTT^{net}_{filtered} < \tau^{NET(k)}_{cutoff} \&\& m^{(k)} == Local$ **then**

  //Set margin to the next cutoff condition

  $margin \leftarrow (1+\theta) \cdot RTT^{net}_{filtered}$

  $\tau^{NET(k+1)}_{cutoff} \leftarrow max(\tau^{NET(k)}_{cutoff}, margin)$

  $m^{(k+1)} \leftarrow Origin$

end

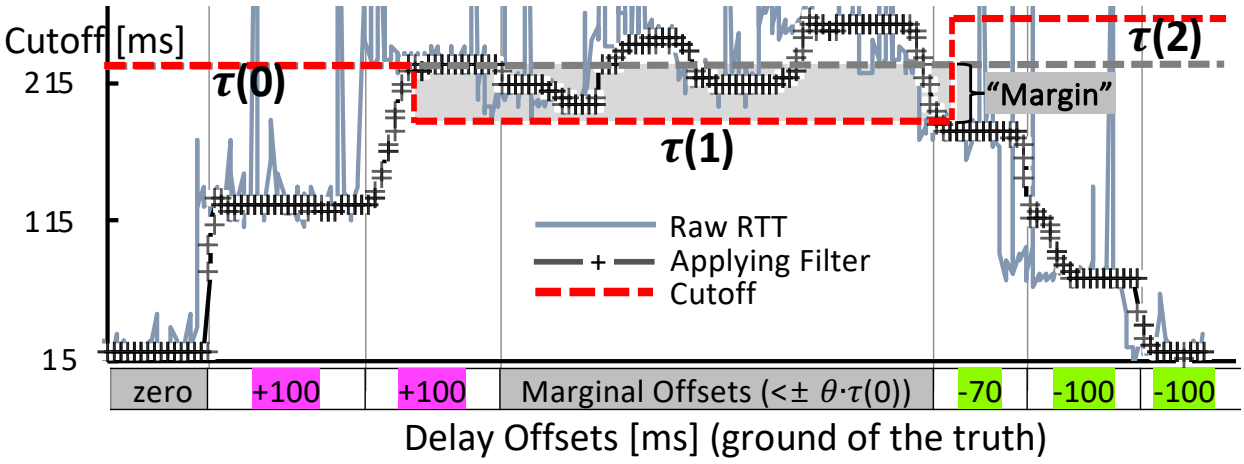
Moving code before reaching a degraded network

Notice that insourcing cannot be accomplished over a limited network. Hence, the procedure needs to be initiated when the network conditions start deteriorating, but before they have reached the point of becoming poor. Since the conditions of a typical mobile network can fluctuate, going up and down, the insourcing commences when the conditions degrade to a given threshold, at which it is still possible to transfer the required source code and state

from the server to the client. After the insourcing, if the conditions deteriorate further, the execution switches into the local mode; however, if they improve, the insourcing is discarded, and the execution continues remotely.

Specifically, to detect the network degradation point, the **CWV** is monitoring successive increases in RTTs, which are larger than $RTT^{NET}_{limited}$, a configurable parameter used to identify if the network is becoming limited[2].

In terms of the actual operations, in the original mode, the app can receive a transmitted code and state from the server at this network degradation point with Insourcing Request. However, this transmission is not applied to the local vessel by `moveToLocalVessel` until the cutoff point.



(a) Updates /bigtrip's cutoff τ on different network conditions ($\tau^{(0)}$=224ms)

Filter	Margin	Mode Switching
✓	✓	**Origin** ↕ **Local** ↕ **Origin**
✓	–	Origin ↕ Loc ↕ Orig ↕ Loc ↕ Orig ↕ Loc ↕ Origin
–	–	Origin ↕↕ ↕↕ ↕↕ ⋯ ↕↕ ↕ ↕ ↕↕ ↕↕ ⋯ ↕↕ ↕↕ ⋯ ↕↕ ↕↕ Origin

(b) Effects of Filter/Margin (θ=0.2) on mode switching for Fig.6.4 (a)'s scenario

Figure 6.4: Monitoring Network Conditions and Adapting Distribution

[2]We set its default value to 4 secs, the ping command's default timeout.

6.2.4 Estimating Network Delay

CWV-enabled mobile clients continuously monitor the underlying network conditions to determine whether to initiate a redistribution to adapt to the latest changes. To assess network conditions, the client collects the RTT_{raw}^{net} metric that represents raw network delay. Specifically, the client is continuously monitoring the RTT_{raw}^{net} by subtracting $T(f_r)$ from $RT(f_r)$, which are obtained from the server.

Since the raw roundtrip is subject to sudden spikes [47], CWV filters out such temporary fluctuations by applying an adaptive filter [69]. Fig. 6.4 shows how applied filter removes the confusing noise. Compare the ground truth and CWV's switches, both the filter and the margin in CWV are important to ascertain the major trends in the changes of network delay.

6.2.5 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 [36].

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.

6.2.6 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. To that end, we *sandbox* the insourced code.

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.

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

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[..]`. Other sandboxing techniques with advanced programming techniques also attenuate the capabilities of accessing web components [66, 75].

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

6.3.1 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 [106]. 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.

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', 'Restify', etc., while for client-side keywords we used 'Ajax', 'Angular', etc. Table 6.1 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 6.1, 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), and Device 2 is an A8-iphone 6 (2 x 1.4GHz); Device 1 outperforms Device 2. The server is an Intel desktop (i7-7700 4 x 3.6 GHz). 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 6.1 demonstrates that the *cutoff latency* of Device 2 (τ^{D2}_{cutoff}) is always larger than that of Device 1 (τ^{D1}_{cutoff}).

Table 6.1: Subject Remote Services for Evaluating CWV

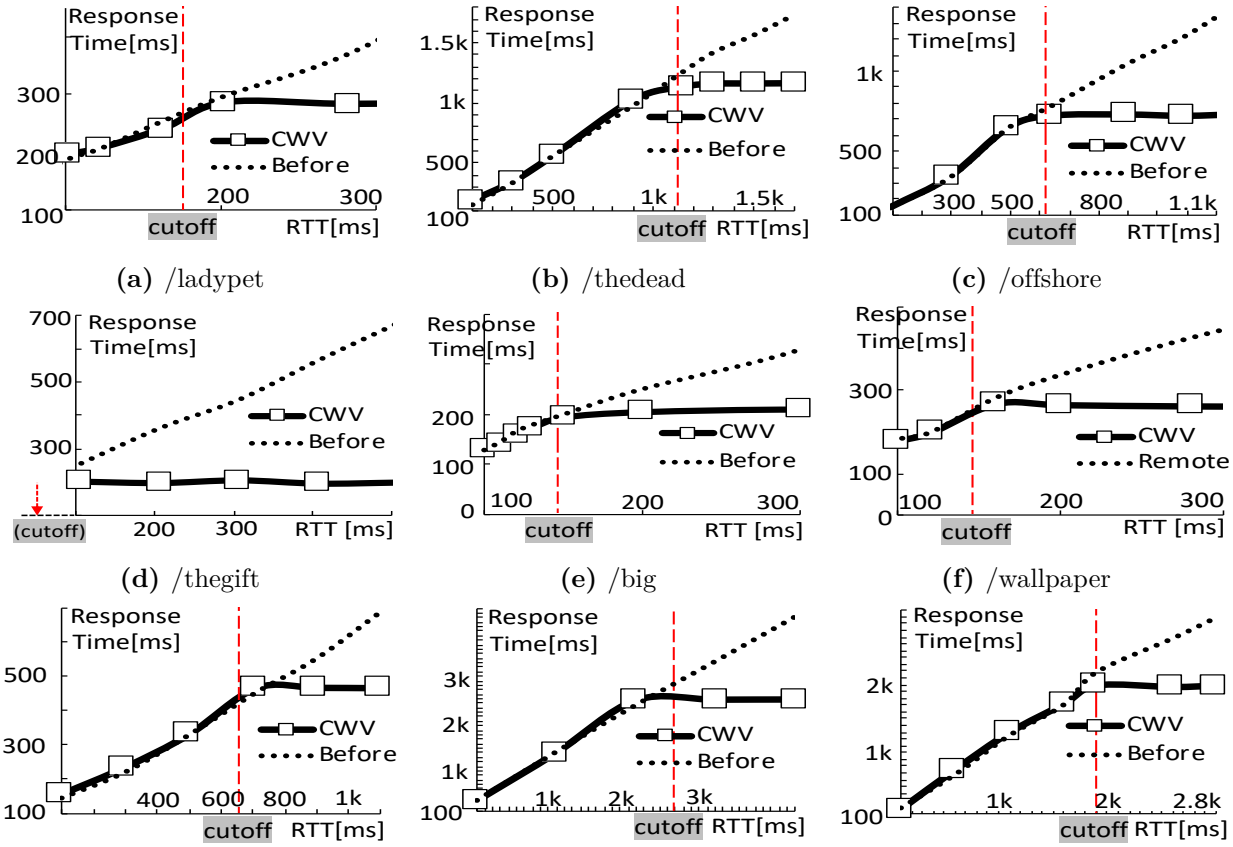
Subject (# of files)	**Remote Services**	$\tau^{\mathbf{D1}}_{\mathbf{cutoff}}$ (msec)	$\tau^{\mathbf{D2}}_{\mathbf{cutoff}}$ (msec)
Bookworm (*729 files*)	/ladypet	176ms	421ms
	/thedea	1120ms	2332ms
	/thered	158ms	424ms
	/thegift	97ms	120ms
	/bigtrip	146ms	224ms
	/offshore	619ms	1528ms
	/wallp	146ms	458ms
	/thecask	90ms	102ms
DonutShop (*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 (*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 (*4k files*)	/user	1.33ms	2.71ms
	/user:id	1.72ms	2.92ms
chem-rules *2.8k files*)	/hbone	26ms	59ms
	/molec	131ms	202ms
benchmark in [106] (*117 files*)			
str-fasta	/str-fasta	656ms	1424ms
fannk	/fannk	2576ms	4982ms
s-norm	/s-norm	1896ms	4873ms

6.3.2 Network Latency Impact

To answer **RQ2**, we set up a test-bed for evaluating network latency impact (See Fig. 8.2-(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 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. 6.5 shows the performance results (The cutoff equals to τ^{D1}_{cutoff} in Table 6.1).

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

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



(a) /ladypet **(b)** /thedead **(c)** /offshore

(d) /thegift **(e)** /big **(f)** /wallpaper

(g) /string-fas **(h)** /fannkuch **(i)** /spectral-n

Figure 6.5: Client's Responsiveness Comparisons

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.

6.3.3 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 power (mW). Fig. 8.2 shows the obtained samples of the 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. 8.2 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 [26]. Even tough, the device switches into the low power mode during the idle states, the longer execution consumes more energy overall.

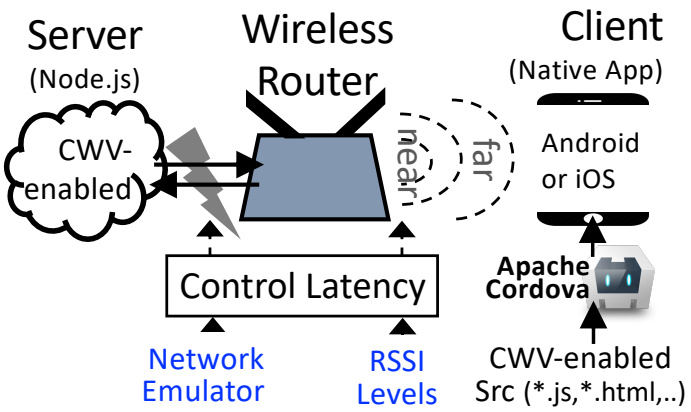
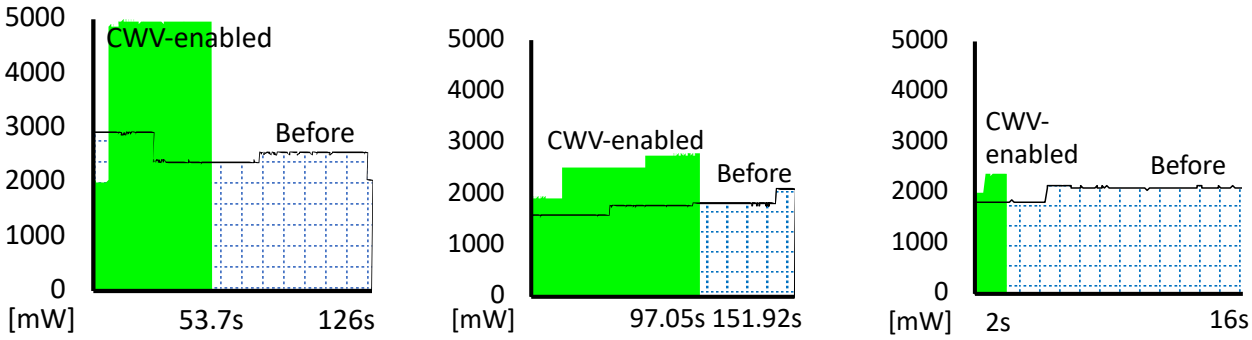


Figure 6.6: Testbed:Latency Control



(a) /str-f (▼74J) **(b)** /thed (▼24J) **(c)** /hbon (▼31J)

Figure 6.7: Testbed and Consumed Energy

6.3.4 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 (**RQ3**), 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

Subjects	Tr_{reg}	Tr_{cwv}	naïve
string-fst	0.383	1.3	374.0
fannkuch	0.367	0.99	375.0
spectral-norm	0.386	1.3	375.2
recipe(4)$^{\Psi}$	0.45	1.1	**13100**
Bookworm(8)	**42.0**	15.8	2412
donuts-shop(4)	0.46	1.1	7542
med-charm(2)	21.9	**395.1**	7542
Average	**22.5**	**54.4**	**4154**

Table 6.2: CWV Overhead for Subjects

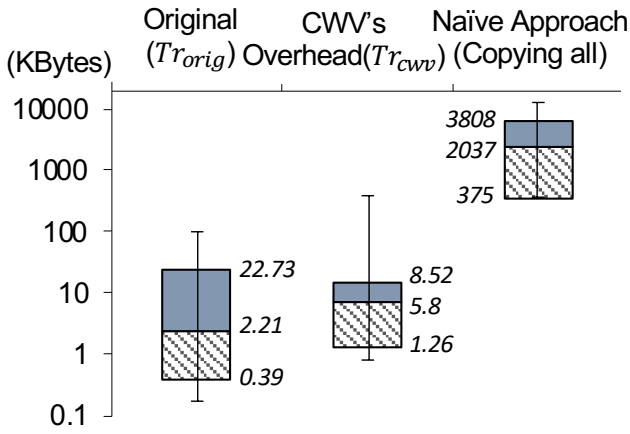


Figure 6.8: CWV's Overhead

needs to replicate about 10K server-side DB entries. However, the transmitting overhead is occurred only once at initialization as these services are stateless. The resulting overall overhead ratio $\frac{Tr_{cwv}}{Tr_{reg}}$ turned out to be **2.4** on average for our subjects (Table 6.2).

To quantify the benefits of **CWV**'s insourcing transferring only the necessary code and state, we also measured the overhead of the so-called naïve approach, which transfers the entire server-side code and state to the client. The performance overhead of transferring everything is about two orders of magnitude slower that **CWV**'s optimized insourcing, an unacceptable slowdown for any practical purposes (Fig. 6.8).

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 $\frac{Tr_{cwv}}{Tr_{reg}}$ turned out to be **2.4** on average for our subjects (Fig. 6.8). 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. 6.8).

6.4 Discussion

The validity of our evaluation is subject to threats and our approach has certain applicability limitations.

6.4.1 Threats to Validity

Our evaluation is subject to both internal and external threats to validity that we discuss in turn next.

Internal Threats

We chose to perform all code analysis and transformation at runtime, even though some of these tasks could have been performed offline. In other words, we could have transformed the client code statically by adding to it the remote functions that might need to be executed

locally. In fact, such static code transformation would allow us to take advantage of the advanced optimization capabilities of modern JavaScript engines that remove the overhead of invoking constructors. Nevertheless, we chose to transform the code at runtime for maximum flexibility at the cost of performance. One can further optimize our implementation by replacing dynamic code migration with a separate post-compilation phase. Hence, our evaluation numbers are reflective of the slowest possible implementation strategy and do not unfairly characterize the efficiency of our approach.

External Threats

We measure the energy consumption of Android devices with the Snap dragon profiler, whose estimation procedure is model-based. If we were to use a power monitor instead, our energy measurements could have differed. Nevertheless, when evaluating energy consumption, our focus is the difference between the local and remote execution of certain functionalities rather than their raw energy consumption numbers. Hence, the obtained energy measurements are sufficient to answer our evaluation questions.

To evaluate how device choice impacts the mode switching points, we used Android and iOS devices. One could argue that the actual execution environment of clients is a mobile browser, whose execution is affected mainly by the underlying device's hardware components rather than the mobile platform. Indeed, as we have observed, the actual cutoff is heavily affected by the device's CPU speed, with the differences stemming from the device's platform being quite modest.

6.4.2 Applicability and Limitations

Our approach's reference implementation works only with JavaScript source code and SQL database code. Nevertheless, some mobile web applications are multilingual (i.e., written in different programming languages and database query languages). Nevertheless, our key ideas and new technologies can be extended for multilingual distributed applications. As automatic language translation has been entrenched into modern software development, integrating mature language translators with our infrastructure is mainly an engineering issue. Furthermore, full-stack JavaScript applications are starting to dominate the development landscape of distributed web applications, as their monolingual nature lowers the development and maintenance burdens, requiring programming proficiency in only JavaScript, used across the development stack.

For various reasons, some remote functionalities cannot be insourced to run on the client, thus making it impossible to create a centralized variant of certain distributed applications. In those cases in which distribution is inevitable, some application resources, naturally remote to the rest of the functionality, cannot change their locality. For instance, news readers display the stories deposited to some centralized repository. It would be impossible to move the news functionality away from the repository to the client, without manually creating some mock components that realistically emulate the appearance of news content locally. In other words, some remote functionalities may depend on resources that cannot be easily migrated away from their host environment for reasons that include relying on server-specific APIs or being dependent on some hard-to-move infrastructure components.

Chapter 7

Related Work

The research presented in this dissertation is broadly related to the areas of Program Transformation and Refactoring, Testing distributed applications, Middleware, Synchronization and Replication, and Cloud and Edge Solutions. We next present the most closely related representatives of related work, comparing and contrasting them with our research.

7.1 Program Transformation and Refactoring

Client Insourcing belongs to a category of refactoring transformations that change the locality of application components for various reasons. One prominent direction in this research is *application partitioning*, which is an automated program transformation that transforms a centralized application into its distributed counterpart [11, 34, 60, 61, 94, 100]. Another approach that leverages compiler-based techniques is the ZØ compiler [34], which automatically partitions CSharp programs into distributed multi-tier applications by applying scalable zero-knowledge proofs of knowledge, with the goal of preserving user privacy.

In JavaScript, choosing one programming implementation over another can make a significant difference performance effect on the overall application. Mainstream of JavaScript Refactoring is debugging JIT (Just-in-time) unfriendly code [37] or removing performance or security problems oriented from bad coding practice [38, 87]. An empirical study identifies reappearing patterns of inefficient JavaScript programs in open source community, so

common performance bottlenecks can be automatically detected and fixed by using software engineering techniques [87]. The JavaScript language constructs for programming event-based applications that wait for dispatches events or message asynchronously. To enhance web-based executions, some advanced static analysis approaches apply formal reasoning for *callback* and *promises* based on a calculus [64, 65]. Declarative program analysis frameworks statically analyze JavaScript code by means of a constraints solver [40, 57, 62, 98].

Client Insourcing can be seen as a variant of program synthesis [12, 28, 29, 30, 31, 32, 42, 92], an active research area concerned with producing a program that satisfies a given set of input/output relationships. CodeCarbonReply [94] and Scalpel [11] integrate portions of a C/C++ program's source in another C/C++ program by leveraging advanced program analysis techniques. The programmer's effort is only limited to annotate the code regions to integrate, and then the tool automatically adapts the receiving application's code to work seamlessly with the moved functionality. However, these works studied how to integrate two independent centralized programs. Client Insourcing synthesizes the distributed version of program to generate its equivalent variant.

7.2 Testing Distributed Applications

Since testing distributed apps defeats static analysis approaches, dynamic analyses have been applied to help test various properties of distributed programs.

Record and Replay(R&R) is an execution framework that efficiently captures distributed execution traces [2, 80]. One of the weaknesses of R&R is its heavy performance overhead due to the need to execute instrumented code over middleware. To reduce this overhead, Parikshan [7] replicates network inputs to remove the need for heavyweight instrumentation by using lightweight containers, thus triggering buggy executions in production with low

overhead. By eliminating distribution altogether, Client Insourcing enables changing in the centralized equivalent of the subjects, thereby providing a low-overhead execution approach.

Another approach is client-side scripting or UI interfacing that automates the client's execution by exercising UI elements in the client-side [50, 68, 71, 72]. Testing a web app requires a sequence of executions. Another limitation of R&R or UI is that one execution can alter the subsequent states of web apps. As a result, testing web apps ends up with false positives or false negatives over test-suits. *Test isolation* makes the testing predictable in the presence of stateful data in web app. Checkpointing execution can automate the isolated execution otherwise programmers should change existing tests to guarantee isolated executions [14, 15, 39].

Profiling frameworks can analyze complex web app structures. MemInsight [48] profiles the memory behavior of DOM objects, and the exact object lifetimes. JSweeter [109] instruments the code patterns related to the type mutation of V8. MultiSE [89] effectively generates testing input values of a JavaScript program by means of dynamic symbolic execution(DSE). Both of the client-side and the server-side applications operate by using asynchronous communication throughout the HTTP middleware.

To capture memory leaks in web applications, BLeak [103], an automated debugging system, identifies memory leaks by checking for a sustained memory growth between consecutive executions. Currently, all these approaches need to be applied separately to the server or client parts of full-stack JavaScript applications because those of profiling frameworks are based in a virtual machine. With Client Insourcing, these approaches becomes immediately applicable for debugging these applications in their insourced versions that execute within a single JavaScript engine.

To handle the large search spaces of possible executions in cloud services, fuzzing techniques

have been applied to automatically manipulate tests by capturing live HTTP requests and then fuzzing the traffic [10]. Restler [9] analyzes swagger specifications and execution feedback to find bugs in cloud-based services in the presence of dependencies across REST APIs. Whereas, Client Insourcing took advantage of fuzzing to identify business logic from the entire server program.

7.3 Middleware

Several middleware-based approach has been proposed to reduce the costs of invoking remote functionalities. APE [78] is an annotation based middleware service for continuously-running mobile (CRM) applications. APE defers remote invocations until some other applications switch the device's state to network activation. Similarly, to reduce the overhead of HTTP communication, events for HTTP requests in Android apps are automatically are bundled into a single batched network transmission [55, 56]. The e-ADAM middleware [53] optimizes energy consumption by dynamically changing various aspects of data transmission scheme. Caching [70] can be particularly beneficial for read-mostly services. If the replicated service data is simply cached, it can then be accessed with low latency. However, caching may be inapplicable as a method for replicating certain remote services. In the presence of state changes, the cached service data can become stale fast. With batching, the proxy server aggregates multiple client requests to forward a single message containing the aggregated data to the server, which also returns the results in bulk [21, 46]. A batching proxy can then reduce the number of WAN transmissions. Batching is most effective in high-bandwidth networks. However, if the amount of transmitted data saturates the available bandwidth, batching becomes ineffective. However, many of stateful services cannot be cached, this proxying strategy's applicability is rather low. Client Insourcing can replicate stateful remote

executions to enhance the responsiveness.

7.4 Distributed Replication and Consistency

As a fault tolerance technique, State Machine Replication (SMR) synchronizes shared states them across distributed machines. To achieve strict ordering, SMR follows a consensus protocol, whose known representative examples include Paxos [54] and Raft [79]. A major shortcoming of SMR are high performance and network communication overheads, required to achieve strong consistency [96]. In contrast, relaxed consistency models incur lower synchronization costs. For this reason, as a consequence, many real-world CRDT-based systems, including Redis and MongoDB, offer relaxed consistency models. Conflict-Free Replicated Data Types (CRDT) [90] is a proven solution for relaxed consistency, a CRDT is a predefined data structure, whose replicas' concurrent updates are eventually synchronized. To achieve state convergence, CRDTs follow mathematically sound update strategies. Due to the relaxed consistency semantics, the replicated state is synchronized in a background process without interfering with the provisioning of main functionalities [93]. The convergence of replicas rendered by CRDT is proved by using automated proof assistant frameworks Isabelle/HOL [36, 36, 49].

7.5 Facilitating Migration to Edge Computing or Centralized Computing

Ours is not the only approach concerned with facilitating the migration of server-side components and data to the edge or the client [22, 85, 92]. Meteor [85], a JavaScript framework,

transparently replicates given parts of a server-side MongoDB database at the client, so these parts can be used for offline operations. Browserify [3] enables a browser to use modules in the same way as regular Node.js modules at the server. WebAssembly [43] provides portable low-level bytecode to execute components written in a variety of programming languages in a browser. WebAssembly has been enhanced with formal type and memory safety guarantees [43, 107]. Servers and browsers execute code in dissimilar ways and browser-based execution of JavaScript code is typically slower than server-based execution. RT.js [25] prioritizes the execution of browser-based real-time jobs within the browser's event queue, so they meet real-time timeliness constraints.

Some approaches migrate database-dependent server code to the client. In modern mobile apps, both the client and server parts may include separate database engines, used for managing persistent data. However, mobile clients and remote servers typically used different database engines, with dissimilar schemas and query interfaces. A data reverse engineering techniques explicitly reconstruct the database schema [20, 22]. Konure [92] infers both the data types and SQL commands of apps interfacing with a database by analyzing execution traces using an active learning approach. Client Insourcing also analyzes database traces, albeit at runtime. Client Insourcing explicitly replicates all table data by instrumenting the identified SQL invocations.

Chapter 8

Future Work

The automated software analysis and transformation toolset created by this dissertation research has a range of applications that can help solve some of the most salient problems faced by the modern computing ecosystem. In this chapter, we will outline some of possible future research directions. One of our current research efforts focuses on integrating edge-based computing and storage resources into existing distributed systems by replicating cloud services at the edge (Section 8.1).

All reference implementations described in this dissertation are implemented under the assumption that they would be applied to monolingual systems, such as full-stack JavaScript applications. However, many modern distributed applications are multilingual, with the client and server parts written in different languages, often quite dissimilar. We describe some possible future work directions that could extend Client Insourcing to work with such multilingual systems (Section 8.2).

8.1 Edge Refactoring

Distributed mobile apps take advantage of cloud services to achieve performance and scalability. Consider a mobile app that collects sensor data to train a machine learning (ML) model. Mobile and IoT devices feature numerous sensors that continuously collect sensor data. This data is passed as input for training ML models. The app then uses the trained

models to optimize and specialize its execution. Since training ML models is a computationally intensive task, the superior and elastic resources of cloud services allow increasing performance at scale.

However, as the number and variety of mobile and IoT devices have been rapidly increasing, so has the volumes of the collected sensor data, referred to as *sensor data deluge* [8, 13]. Due to this development, it is no longer viable to transfer all the collected sensor data to the cloud for processing. The resulting network transmission bottlenecks would negate all performance advantages offered by using superior cloud-based computing resources. As a solution to this problem, edge computing has been explored as an enhancement of traditional cloud computing. In this distributed architecture, local computing resources at the edge of the network process much of the collected sensor data, thus avoiding the necessity to transfer large volumes of data over WANs and the associated performance bottlenecks.

Figure 8.1 depicts the dataflow of a third-party distributed mobile app, *firebase-objdet-node*, which is distributed across a mobile client and a cloud server. Specifically, the client captures images and sends them to the server for processing. The server analyzes the received images, returning the analysis results back to the client. To that end, the server performs computationally intensive operations that localize and identify the constituent objects in the received images, as guided by the pre-trained model of a deep learning framework. The server then transfers the results to the client, which uses them to draw the boundaries of and descriptions around the identified objects in the captured images. Assume that the app is deployed for a mission-critical task, such as security monitoring, so it is essential to achieve the requisite levels of response latency.

Modern smartphones typically come with 8-16 megapixels cameras. An image captured with such cameras would typically take between 1 and 20 Mbytes. As images are being transferred to the cloud-based image processing service, the network's bandwidth and latency

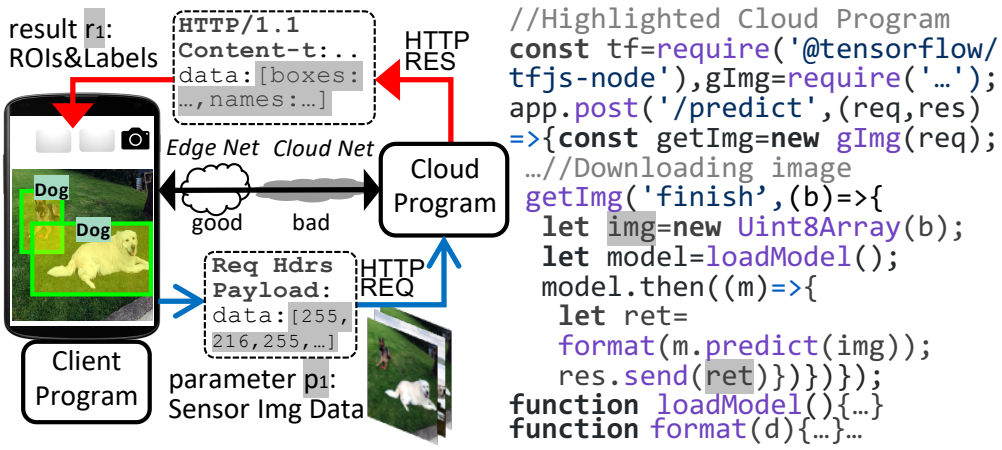


Figure 8.1: Motivating Example: *firebase-objdet-node*

determine the resulting performance. In addition, providers of cloud services can host them in different geographic regions, not necessarily those co-located with the client. In fact, the actual geographic of the service can drastically affect the round-trip time (RTT) metric. To demonstrate this insight, we installed our example app's remote service on the cloud infrastructures, located on the same continent and on the nearest neighboring continent. The RTT across different continents is *an order of magnitude larger* than within the same continent. Because the app is mission-critical, experiencing such a slowdown in performance would cause the app to fail in its mission.

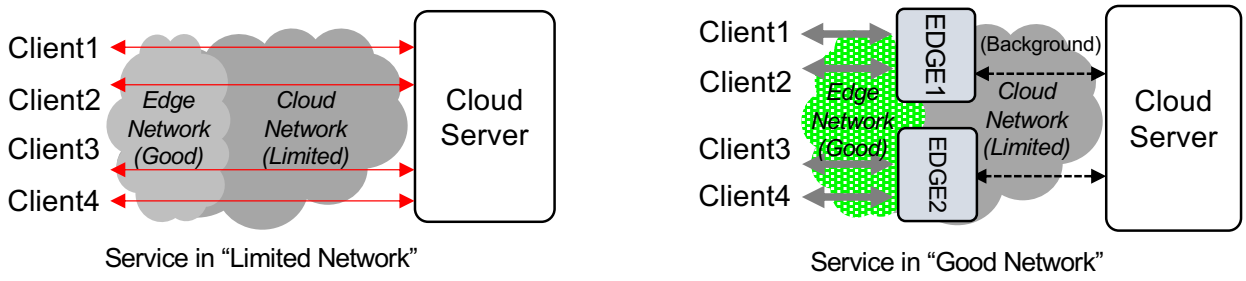
To work around the network bottleneck conditions described above, one can take advantage of edge computing resources, provided by nearby network-connected computing devices. Such devices can be accessed in a single hop within the shared local network, but would offer less processing power than their cloud-based counterparts. In other words, some image processing can be offloaded to nearby edge devices for processing, so the app would adaptively take advantage of edge computing resources for certain combinations of processing loads and RTTs.

To take advantage of edge resources, new applications can be developed from scratch. How-

ever, a more common scenario is when developers decide to introduce edge processing to existing cloud applications as a performance optimization. In other words, developers end up modifying existing cloud-based applications to integrate edge-based processing. In essence, developers transform two-tier (device-cloud) applications into three-tier (device-edge-cloud) applications. To transform such an app correctly by hand, so its performance would improve as intended, is non-trivial. Developers first have to understand the app's distributed runtime behavior in order to identify those cloud-based functionalities that can benefit from edge-based processing. Then, developers have to determine whether these functionalities maintain state and how to keep it consistent once replicated. Finally, developers have to modify by hand distributed communication protocols, commonly implemented by means of special frameworks with complex APIs.

Our solution comprises a behavior-preserving automated program transformation, implemented as the EDGE REFACTORING framework. EDGE REFACTORING attaches to a running client-cloud application and captures its live HTTP traffic between the client and the invoked cloud-based services. Based on the captured traffic, EDGE REFACTORING then analyzes and transforms the original two-tier application to a semantically equivalent three-tier application. The transformed application continues delivering the original functionality, but with improved latency and throughput. The transformation dynamically generates and instantiates a *Remote Proxy*: the edge node becomes the cloud server's filtering and processing proxy, preserving the original service interfaces. As a high level overview, certain functionality of a cloud-based services becomes replicated at edge nodes, co-located with clients. The service states are then synchronized between the cloud and the edge replicas in the background.

One could simply duplicate the entire server-side functionality at the edge nodes, synchronizing all of the replicated service state. However, such a naïve approach would incur unaccept-



(a) 2-tiers Architecture(Cloud-Client) **(b)** 3-tiers Architecture(Cloud-Edge-Client)

Figure 8.2: Edge Refactoring

ably high synchronization costs, so the amount of functionality to transfer to and execute by the edge replicas must be carefully selected. In addition, failure handling logic may not be easily transferred from the cloud to the edge, and may require managing complex states. To allow for easy replication at the edge, the edge replicas handle failures by forwarding the failed service invocations to the master service copy in the cloud. This scheme reduces the overheads and complexity of not only handling failure but also of synchronizing states.

8.2 Transitioning from Partial to Full-Stack JS Apps

A growing number of enterprises are increasingly adapting full-stack JavaScript application as their preferred implementation architecture for web applications. The reasons for why enterprises find this architecture enticing are quite obvious. Its monolingual nature reduces the development complexity and costs, making it easier to find qualified developers. That is, the client and server developers can share the same set of development tools (e.g.. IDEs, testing frameworks, linters, etc.), so their roles can become more interchangeable. It is also easier to hire competent developers who are expected to demonstrate proficiency only in JavaScript and its development infrastructure.

There is great potential benefit in transforming partial-stack JavaScript applications, in

which only the client is written in JavaScript, into full-stack JavaScript application to reap all the aforementioned benefits. However, the transitioning from partial-stack to full-stack JavaScript applications is a laborious and error-prone undertaking. First, this transitioning requires translating the server functionality written in Java to JavaScript. Before this language-to-language translation can take place, however, the slice of the code that implements business logic needs to be separated from the middleware API calls for communication and database, with the business logic and middleware closely intertwined.

```
1  import service.*;
2  import org.springframework.*;
3  @RestController
4  public class UserController {
5   private final TaskService service;
6   @GetMapping("/task/{taskname}")
7   void List<Task> getTasks(@PathVariable String name){
8     List<Task> tasks = taskService.findByName(name);
9     if (task.length == 0)
10      return ResponseEntity.notFound().build();
11    return tasks;
12  }
13
14  @PutMapping("task/{id}")
15   public void editTask(@RequestBody Task t,
         @PathVariable("id") Long id){
16     t.setId(id);
17     tasksRepository.saveAndFlush(t);
18     return ResponseEntity.ok().build();
19   }//business logic
20 }
```

```
1  class TaskService extends
       JpaRepository<Task,
       Long>{
2    Collection<Tasks>
         findBytaskname(String
         taskname);
3  ..}//Query Models

1  @Entity//Data Models
2  class Tasks{..}

1  //client part
2   $.ajax({
3   url: '/task/',
4   data: {id: $('#id').val(),
5      ...},//client-input
6   type: 'GET',
7   success: function(data)
        {//serv-output
8   $('#results').text(..)}});
```

Figure 8.3: Motivating partial-stack App:*TODO-List*

Figure 8.3 highlights code snippets for motivating partial-stack application `TODO-List`, it comprises of server and client part written Java and JavaScript, respectively. To make a full-stack app, the server part of the app should be translated by JavaScript with Node.js. By encapsulating all application concerns related to distributed communication and database, middleware streamlines the implementation of distributed applications and has become a

mainstay of modern software development practices. To be specific, the underlined server middleware framework is the *spring framework*, which is the most popular for Java developer's community for over a decade. Java Reflection with annotations is a critical attribute to simplify its programming models for both of communication behaviors and database table retrievals in this framework.

Approach #1: Applying Transpilation

Transpiler uses the source code of a program written in a programming language as its input and generates a corresponding source code in the identical or a different programming language. Transpiling frameworks are used for supporting multiple different programming language platforms and reduce a manual effort of rewriting the source code. To translate from source to target programming languages, transpiling frameworks manually define the rules [27, 82, 97, 110] or automatically infer translation rules from the source codes [1, 6, 35].

For an automated transitioning from Java to JavaScript, developers may think of applying state-of-art transpiling tools. The main problem is that middlewares in multi-tier distributed programs impede seamless transitions because of the programming model for middleware completely differs from each other (i.e., Java Reflection). Translations are ended up with undefined errors for translating mappings between Java and JavaScript. For instance, a state-of-art transpiler for Java-to-JavaScript, JSweet [82] raises translating errors. However, transpilation is still valuable since the transpiler can translate other business logic (i.e., none-middleware parts) into the target language without manual effort.

$$App_{dist} \rightarrow \textsc{Insourcing} \rightarrow \begin{matrix} App^{local}_{server} \\ App^{JS}_{client} \end{matrix} \rightarrow \textsc{Transpiler} \rightarrow App^{JS}_{cent} \rightarrow \begin{matrix} Adapting \\ Enhancing \end{matrix}$$

Approach #2: Applying Universal Virtual Machine

Universal virtual machines enable interoperability between different programming languages in a shared runtime [102, 108]. Universal virtual machines can be used to obviate the necessity for source-to-source translation when insourcing code in applications, whose remote parts are implemented in different languages. The insourcing refactoring would proceed as usual by generating a centralized variant, albeit in the language different from that of the client App^{local}_{server} (e.g, written in Java). The insourced part would remain untranslated, in the language used by the server. But the virtual machine would still make it possible to invoke local functions in the client environment, such as the mobile browser. To enhance or adapt the variant, programmers can apply existing refactoring frameworks that work with the server-side language (e.g., Java refactoring frameworks).

$$App_{dist} \rightarrow \textsc{Insourcing} \rightarrow \begin{matrix} App^{local}_{server} \\ App^{JS}_{client} \end{matrix} \rightarrow \textsc{Universial VM} \rightarrow App^{JS}_{cent} \rightarrow \begin{matrix} Adapting \\ Enhancing \end{matrix}$$

GraalVM [108] is a universal virtual machine developed by Oracle Labs. GraalVM supports communication between objects, implemented in different languages. Another approach that supports language interoperability is Protocol Buffers [83], which serializes objects across languages and then shares the data by means of its language-independent specification. Polyfills [91] provides adapters to support the communication between objects implemented in different languages. It also make it possible for the same code to be executed on different client platforms.

Chapter 9

Conclusion

The ever-changing realities of modern distributed apps put new obstacles on the road of providing a satisfactory user experience. In particular, modern users expect distributed applications to be responsive, reliable, and energy efficient. Distributed application developers need powerful approaches, techniques, and tools that allow them to reach these objectives on time and under budget.

This dissertation research has innovated in the software engineering space to create novel automated reengineering approaches that enhance and optimize distributed execution. In particular, this dissertation has introduced a novel domain-specific refactoring, supported by state-of-the-art program analysis and transformation techniques, and applied this refactoring to address some of the most salient problems of distributed execution. The novel approaches and techniques introduced by this dissertation can inform other efforts that aim at improving the state of the art in maintaining and evolving distributed applications. It is our hope that our collective efforts would lead to the creation of the next generation of software reengineering tools that have strong potential for practical adoption.

Bibliography

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] Gautam Altekar, Ion Stoica, Gautam Altekar, and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *In SOSP*, 2009.

[3] Tim Ambler and Nicholas Cloud. Browserify. In *JavaScript Frameworks for Modern Web Dev*, pages 101–120. Springer, 2015.

[4] Kijin An. Facilitating the evolutionary modifications in distributed apps via automated refactoring. In *Web Engineering*, pages 548–553. Springer International Publishing, 2019. ISBN 978-3-030-19274-7.

[5] Kijin An and Eli Tilevich. Client insourcing: Bringing ops in-house for seamless re-engineering of Full-Stack JavaScript Applications. In *Proceedings of The Web Conference 2020*, pages 179–189, 2020.

[6] Kijin An, Na Meng, and Eli Tilevich. Automatic inference of java-to-swift translation rules for porting mobile applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, page 180–190, 2018. ISBN 9781450357128.

[7] Nipun Arora, Jonathan Bell, Franjo Ivančić, Gail Kaiser, and Baishakhi Ray. Replay without recording of production bugs for service oriented applications. In *Proceedings*

of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 452–463. ACM, 2018.

[8] Marcos D Assunção, Rodrigo N Calheiros, Silvia Bianchi, Marco AS Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79:3–15, 2015.

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[10] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498*, 2020.

[11] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, 2015. ISBN 978-1-4503-3620-8.

[12] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, volume 52, pages 95–110. ACM, 2017.

[13] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919): 1297–1298, 2009.

[14] Jonathan Bell. Detecting, isolating, and enforcing dependencies among and within test cases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 799–802, 2014.

[15] Jonathan Bell and Gail Kaiser. Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, pages 550–561, 2014.

[16] Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[17] FP Brooks Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[18] Eric J Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the Conference on Software Maintenance*, pages 226–235, 1992.

[19] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[20] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. Dynamic program analysis for database reverse engineering. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 297–321. Springer, 2011.

[21] William R Cook and Ben Wiedermann. Remote batch invocation for sql databases. In *DBPL*, 2011.

[22] Kathi Hogshead Davis and PH Alken. Data reverse engineering: A historical survey. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 70–78. IEEE, 2000.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[25] Christian Dietrich, Stefan Naumann, Robin Thrift, and Daniel Lohmann. Rt.js: Practical real-time scheduling for web applications. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 69–79. IEEE, 2019.

[26] Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y Charlie Hu, and Andrew Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 29–40. ACM, 2013.

[27] M. El-Ramly, R. Eltayeb, and H. A. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, 2006.

[28] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, pages 1297–1305, 2016.

[29] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *ACM SIGPLAN Notices*, volume 52, pages 422–436. ACM, 2017.

[30] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Notices*, volume 53, pages 420–435. ACM, 2018.

[31] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[32] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[33] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[34] Matthew Fredrikson and Benjamin Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 909–924, 2014. ISBN 978-1-931971-15-7.

[35] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring likely mappings between APIs. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 82–91. IEEE, 2013.

[36] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[37] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, 2015. ISBN 978-1-4503-3675-8.

[38] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically

checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, 2015.

[39] Marco Guarnieri, Petar Tsankov, Tristan Buchs, Mohammad Torabi Dashti, and David Basin. Test execution checkpointing for web applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 203–214, 2017.

[40] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, pages 78–85, 2009.

[41] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[42] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[43] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[44] Michael Hilton, Arpit Christi, Danny Dig, Michał Moskal, Sebastian Burckhardt, and Nikolai Tillmann. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pages 83–92, 2014.

[45] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *ECOOP 2009 – Object-Oriented Programming*, pages 595–617, 2009. ISBN 978-3-642-03013-0.

[46] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R Cook. Remote batch invocation for compositional object services. In *European Conference on Object-Oriented Programming*, pages 595–617. Springer, 2009.

[47] Krister Jacobsson, Håkan Hjalmarsson, Niels Möller, and Karl Henrik Johansson. Estimation of rtt and bandwidth for congestion control applications in communication networks. In *IEEE CDC, Paradise Island, Bahamas*. IEEE, 2004.

[48] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[49] Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. Designing a planetary-scale imap service with conflict-free replicated data types. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[50] Emre Kiciman and Benjamin Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *ACM SIGOPS Operating Systems Review*, pages 17–30. ACM, 2007.

[51] Y. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 586–595, 2012. doi: 10.1109/ICDCS.2012.75.

[52] Y. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 586–595, 2012.

[53] Young-Woo Kwon and Eli Tilevich. Configurable and adaptive middleware for energy-efficient distributed mobile computing. In *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services (MobiCASE)*, pages 106–115. IEEE, 2014.

[54] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[55] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. An empirical study of the energy consumption of Android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 121–130. IEEE, 2014.

[56] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. Automated energy optimization of HTTP requests for mobile applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 249–260. IEEE, 2016.

[57] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014. ISBN 978-1-4503-3056-5.

[58] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978. ISSN 0001-0782. doi: 10.1145/359511.359522. URL https://doi.org/10.1145/359511.359522.

[59] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017*

IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 399–410, 2017.

[60] Yin Liu, Kijin An, and Eli Tilevich. RT-Trust: Automated refactoring for trusted execution under real-time constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, pages 175–187. ACM, 2018. ISBN 978-1-4503-6045-6.

[61] Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: Automated refactoring for different trusted execution environments under real-time constraints. *Journal of Computer Languages*, page 100939, 2019.

[62] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[63] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, 2014. ISBN 978-1-4503-3056-5.

[64] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, 2015.

[65] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, (OOPSLA): 86:1–86:24, October 2017. ISSN 2475-1421.

[66] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted javascript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91. IEEE, 2009.

[67] Josip Maras, Jan Carlson, and Ivica Crnkovi. Extracting client-side web application code. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 819–828, 2012. ISBN 978-1-4503-1229-5.

[68] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of AJAX web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 121–130, April 2008. doi: 10.1109/ICST.2008.22.

[69] Reiner Marchthaler and Sebastian Dingler. *Kalman-Filter*, volume 30. Springer, 2017.

[70] Jhonny Mertz and Ingrid Nunes. Understanding application-level caching in web applications: a comprehensive introduction and survey of state-of-the-art approaches. *ACM Computing Surveys (CSUR)*, 50(6):1–34, 2017.

[71] Ali Mesbah and Arie Van Deursen. A component-and push-based architectural style for AJAX applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.

[72] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling AJAX by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 122–134. IEEE, 2008.

[73] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 11–11, 2010.

[74] Marija Mikic-Rakic and Nenad Medvidovic. A classification of disconnected operation techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 144–151. IEEE, 2006.

[75] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized javascript. *Google, Inc., Tech. Rep*, 2008.

[76] Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel. Specification and detection of SOA antipatterns. In *International Conference on Service-Oriented Computing*, pages 1–16. Springer, 2012.

[77] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 496–499, 2011. ISBN 978-1-4503-0443-6.

[78] Nima Nikzad, Octav Chipara, and William G Griswold. Ape: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 515–526. ACM, 2014.

[79] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[80] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[81] David A Patterson. Latency lags bandwith. *Communications of the ACM*, 2004.

[82] Renaud Pawlak. Jsweet: Insights on motivations and design. *A transpiler from Java to JavaScript. EASYTRUST*, 16, 2015.

[83] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 261–265. IEEE, 2016.

[84] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1996.

[85] Josh Robinson, Aaron Gray, and David Titarenco. Getting started with meteor. In *Introducing Meteor*, pages 27–41. Springer, 2015.

[86] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring*, pages 1–4. ACM, 2016.

[87] M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[88] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, 2013.

[89] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint*

Meeting on Foundations of Software Engineering, ESEC/FSE 2015, 2015. ISBN 978-1-4503-3675-8.

[90] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[91] Remy Sharp. What is a polyfill. *Remy Sharp*, 2010.

[92] Jiasi Shen and Martin C Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–285. ACM, 2019.

[93] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[94] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 95–105, 2017.

[95] Jon Siegel and Dan Frantz. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000.

[96] Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable state machine replication of state-based CRDTs without logs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 455–457, 2019.

[97] Harry M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010.

[98] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static dom event dependency analysis for testing web applications. In *Proceedings of the 2016 24th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 447–459, 2016.

[99] E. Tilevich and Y. Kwon. Cloud-based execution to improve mobile application energy efficiency. *Computer*, 47(1):75–77, Jan 2014. ISSN 0018-9162. doi: 10.1109/MC.2014.6.

[100] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, pages 178–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-47993-2.

[101] Vassilios Tsaoussidis, Hussein Badr, Xin Ge, and Kostas Pentikousis. Energy/throughput tradeoffs of tcp error control strategies. In *Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications*, pages 106–112. IEEE, 2000.

[102] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *arXiv preprint arXiv:1803.10201*, 2018.

[103] John Vilk and Emery D Berger. Bleak: automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–29. ACM, 2018.

[104] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64. Springer, 1996.

[105] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. 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, pages 83–84, 2012.

[106] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. 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*, pages 83–84. ACM, 2012.

[107] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 53–65, 2018.

[108] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[109] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. Uncovering JavaScript performance code smells relevant to type mutations. In *Asian Symposium on Programming Languages and Systems*, pages 335–355, 2015.

[110] K. Yasumatsu and N. Doi. SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 1995.

[111] Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, 2008.

[112] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic

battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.