Catch & Release: An Approach to Debugging Distributed Full-Stack JavaScript Applications

Kijin An(✉) and Eli Tilevich

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

Abstract. Localizing bugs in distributed applications is complicated by the potential presence of server/middleware misconfigurations and intermittent network connectivity. In this paper, we present a novel approach to localizing bugs in distributed web applications, targeting the important domain of full-stack JavaScript applications. The debugged application is first automatically refactored to create its semantically equivalent centralized version by gluing together the application's client and server parts, 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. We show how our approach can be used to catch bugs that include performance bottlenecks and memory leaks. These results indicate that our debugging approach can facilitate the challenges of localizing and fixing bugs in web applications.

Keywords: Full-stack JavaScript applications · Distributed computing · Debugging

1 Introduction

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

© Springer Nature Switzerland AG 2019

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.

The rest of this paper is structured as follows. Section 2 discusses the state of the art in software debugging. Section 3 introduces our approach for debugging full-stack JavaScript applications. Section 4 presents the design and implementation details of CandoR. Section 5 reports on the evaluation results of applying CandoR to debug real-world subject applications. Section 6 presents future work directions and concluding remarks.

2 Background and Related Work

Numerous prior approaches have pursued the goal of improving the effectiveness and lowering the cognitive burden of software debugging. It would be unrealistic to discuss all of them here. Hence, we outline only some major prior efforts, particularly those that had introduced some of the techniques we used to implement our approach. An important part of the debugging process is exercising the runtime behaviour of the debugged subject. When it comes to testing web applications, client-side scripting and UI interfacing have been introduced to automatically exercise UI elements and to conduct state-based testing [8,10,12,13]. These approaches approximate server-side application logic as simple states. In contrast, CandoR first transforms a distributed application into its equivalent centralized version, in which the original server-side logic is encapsulated in regular functions that can be debugged by applying any of these prior approaches.

To debug distributed applications that execute over middleware, Record and Replay (R&R) is an execution framework that efficiently captures distributed execution traces [1,14]. 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 [2] 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, CandoR enables localizing bugs in the centralized equivalent of the debugging subjects, thereby providing a low-overhead debugging approach.

Since JavaScript defeats static analysis approaches, dynamic analyses have been applied to help understand various properties of JavaScript programs, including performance and memory consumption. MemInsight [7], a profiling framework, can analyze complex JavaScript programming constructs, the memory behavior of DOM objects, and the exact object lifetimes. Dynamic analysis has also been used to identify promising refactoring opportunities in JavaScript code, such as detecting Just-In-Time (JIT)-unfriendly code sections that can be restructured to improve performance. To understand how prevalent JIT-unfriendly code is, JITProf [5] applies dynamic analysis to help developers detect such code regions. To help identify harmful JavaScript coding practices, DLint [6] features a dynamic checker based on formal descriptions. To detect performance bottlenecks, JSweeter [21] analyzes the code patterns related to the type mutation of the V8 engine.

To detect memory leaks in web applications, BLeak [20], 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. With CandoR, these approaches becomes immediately applicable for debugging these applications in their insourced versions that execute within a single JavaScript engine.

3 Debugging Full-Stack JavaScript Applications with CandoR

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.

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

//inefficient for-in loop in server part
     , lines are from 5 to 18
function getObjsInArray(obj, array) {
 var foundObjs = [];
  for (var i=0; i < array.length; i++){
   for(var prop in obj) {
    if(obj.hasOwnProperty(prop)) {
     if (obj[prop] === array[i][prop]) {
      foundObjs.push(array[i]);
      break;
  }}}}
 return foundObjs;
}
```

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

```
//patch for server part

5,18c1,16
<(original code for getObjsInArray)
---
> function getObjsInArray(obj, array) {
>  var foundObjs = [];
>  var keys = Object.keys(obj);
>  for (var i=0; i < array.length; i++)
     {
>   for (var j = 0, l = keys.length; j <
      l; j++) {
>    var key = keys[j];
>    if (obj[key] === array[i][key]) {
>     foundObjs.push(array[i]);
>     break;
>  }}}
>  return foundObjs;
> }
```

Fig. 1. Distributed App theBrownNode (left:server part, upper right:client part) and patch for inefficient iteration. (shaded)

3.1 Motivating Example I: *Removing Performance Bottlenecks*

Consider the code snippet in Fig. 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()` [17].

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. 1) to be executed against the original server or client part of the distributed application (i.e., the "release" phase).

3.2 Motivating Example II: *Detecting Memory Leak*

Some of the most common bugs afflicting remote services are memory leaks. Consider function `leakingService` in Fig. 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 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

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

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

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

Fig. 2. Memory leak examples for server and client parts

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* [20]. 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[1]. 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.

[1] Angular HTTPClient (https://angular.io/guide/http),
JQuery AJAX (http://api.jquery.com/jquery.ajax/),
Node.js HTTP module (https://nodejs.org/api/http.html).

4 CANDOR: Design and 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.

4.1 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. 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 [4], similarly to other declarative program analysis frameworks [9,18].

```
//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));
```

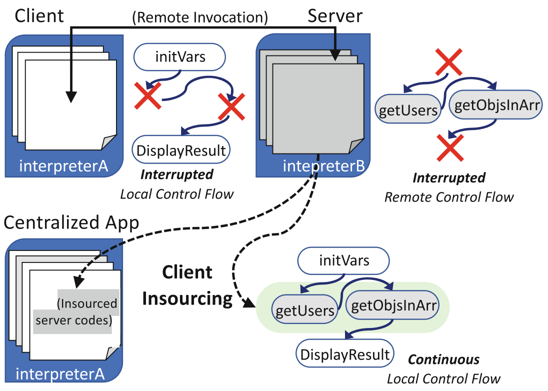


Fig. 3. Continuous control flow of distributed codes (theBrownNode in Fig. 1) constructed by Client Insourcing (left: generated code for centralized applications)

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. 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 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. 3).

4.2 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 [5,7,17]. For example, reference [17] 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* [22] 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 [11] to the debugged application's centralized version. As shown in the Fig. 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.

4.3 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 transformations. 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).

[2] For additional implementation details, see https://bit.ly/2B9a3wf.
[3] https://www.gnu.org/software/diffutils.
[4] http://savannah.gnu.org/projects/patch.

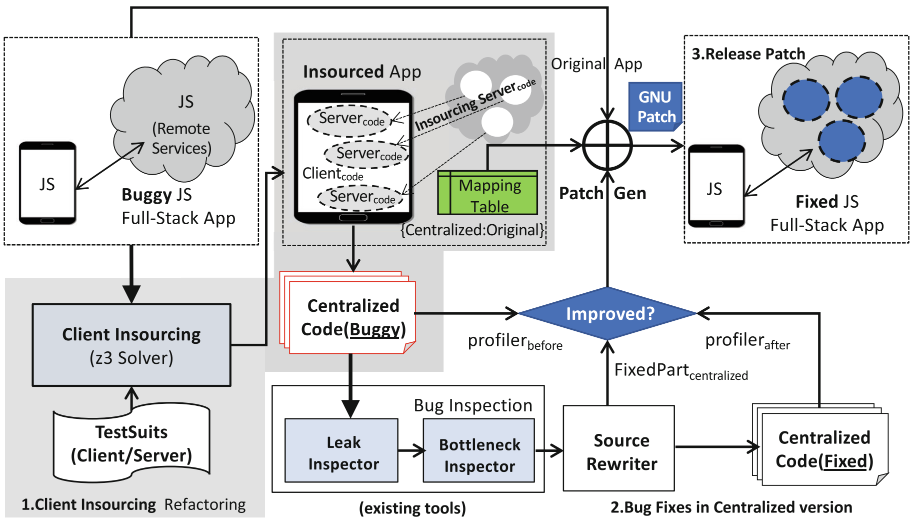


Fig. 4. Debugging full-stack JavaScipt applications with CANDOR

5 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? (Sect. 5.1)
- **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? (Sect. 5.2) How much programmer effort can CANDOR save? (Sect. 5.3)

5.1 Evaluating the Correctness of Client Insourcing

Table 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 1. Subject distributed apps and Client Insourcing results

Subject Apps	S_{ULOC}	C_{ULOC}	Remote services	CI_{ULOC}
theBrownNode [19]	147	43	/users/search	37
			/users/search/id	36
Bookworm [3]	371	1814	/api/ladywithpet	394
			/api/thedea	394
			/api/theredroom	394
			/api/thegift	394
search_leak [16]	34	13	/search_leak	17
ionic2_realty_rest [15]	453	387	/properties/favorites	24

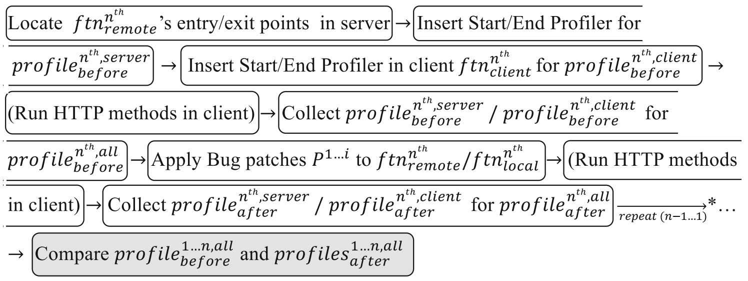
Table 1 shows the total number of tests in each evaluated test suite, including the number of tests manually transformed to work against the centralized versions of subject applications; the table shows whether tests successfully passed in the original and refactored version of each subject. Based on 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.

5.2 Case Study: Traditional vs. CandoR-Enabled Debugging

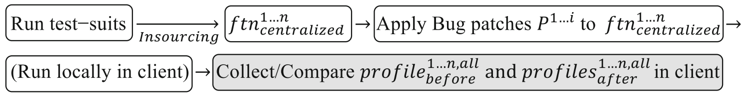
In this case study, we compare and contrast a traditional approach to localizing a bug in a full-stack JavaScript application and the CandoR debugging approach. In this case study, we assume that a programmer needs to debug a distributed application with n remote functionalities[5] $ftn^{1...n}_{remote}$ 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 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

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

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. 5a).


Locate $ftn^{n^{th}}_{remote}$'s entry/exit points in server
Insert Start/End Profiler for $profile^{n^{th},server}_{before}$
Insert Start/End Profiler in client $ftn^{n^{th}}_{client}$ for $profile^{n^{th},client}_{before}$
(Run HTTP methods in client)
Collect $profile^{n^{th},server}_{before}$ / $profile^{n^{th},client}_{before}$ for $profile^{n^{th},all}_{before}$
Apply Bug patches $P^{1...i}$ to $ftn^{n^{th}}_{remote}/ftn^{n^{th}}_{local}$
(Run HTTP methods in client)
Collect $profile^{n^{th},server}_{after}$ / $profile^{n^{th},client}_{after}$ for $profile^{n^{th},all}_{after}$
$\xrightarrow[repeat\ (n-1...1)]{}$ *...
Compare $profile^{1...n,all}_{before}$ and $profiles^{1...n,all}_{after}$

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


Run test–suits
$\xrightarrow[Insourcing]{}$
$ftn^{1...n}_{centralized}$
Apply Bug patches $P^{1...i}$ to $ftn^{1...n}_{centralized}$
(Run locally in client)
Collect/Compare $profile^{1...n,all}_{before}$ and $profiles^{1...n,all}_{after}$ in client

(b) Streamlined Debugging Process with CandoR

Fig. 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. 5b.

5.3 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 2 presents the performance and memory consumption improvements for these debugging 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 $P_{improved} = \frac{P_{before} - P_{after}}{P_{before}} \cdot 100(\%)$. 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 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}).

5.4 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 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.

Table 2. Quantifying debugged results by CANDOR

Remote services	Bug types [17]	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	0.36 ms (0.19 ms)	0.26 ms (0.13 ms)	27.8% (31.58%)
/users/search/id	Inefficient iteration	31	1.7 ns (2.5 ns)	1.19 ns (1.63 ns)	29.53% (34.8%)
/api/ladywithpet	Misused APIs	18	5.89 ms (2.74 ms)	4.99 ms (2.24 ms)	15.28% (18.13%)
/api/thedea	Misused APIs	18	5.63 ms (2.71 ms)	4.82 ms (2.25 ms)	14.39% (16.97%)
/api/theredroom	Misused APIs	18	0.65 ms (1.87 ms)	0.53 ms (1.56 ms)	18.06% (16.58%)
/api/thegift	Misused APIs	18	1.17 ms (0.36 ms)	1.04 ms (0.31 ms)	11.11% (13.89%)
/search_leak	Memory leak	24	619.10 kb (476.16 kb)	519.13 kb (409.10 kb)	16.15% (14.08%)
/properties/favorites	Memory leak	42	824.62 kb (1431.28 kb)	511.37 kb (922.51 kb)	37.99% (35.54%)

6 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.

Acknowlegements. The authors would like to thank Yin Liu and the anonymous reviewers, whose insightful comments helped improve the technical content of this paper. This research is supported by the NSF through the grants # 1650540 and 1717065.

References

1. Altekar, G., Stoica, I., Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: SOSP (2009)
2. Arora, N., Bell, J., Ivančić, F., Kaiser, G., Ray, B.: Replay without recording of production bugs for service oriented applications. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 452–463. ACM (2018)
3. Bookworm. https://github.com/davidwoodsandersen/Bookworm
4. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

5. Gong, L., Pradel, M., Sen, K.: JITprof: Pinpointing JIT-unfriendly JavaScript code. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 357–368. ESEC/FSE 2015 (2015)
6. Gong, L., Pradel, M., Sridharan, M., Sen, K.: Dlint: dynamically checking bad coding practices in JavaScript. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 94–105. ISSTA 2015 (2015)
7. Jensen, S.H., Sridharan, M., Sen, K., Chandra, S.: MemInsight: platform-independent memory debugging for JavaScript. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 345–356. ACM (2015)
8. Kiciman, E., Livshits, B.: AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In: ACM SIGOPS Operating Systems Review, pp. 17–30. ACM (2007)
9. Livshits, B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium, vol. 14 (2005)
10. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of AJAX web applications. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 121–130, April 2008
11. Memwatch. https://github.com/eduardbcom/node-memwatch
12. Mesbah, A., Bozdag, E., Van Deursen, A.: Crawling AJAX by inferring user interface state changes. In: Eighth International Conference on Web Engineering. ICWE 2008, pp. 122–134. IEEE (2008)
13. Mesbah, A., Van Deursen, A.: A component-and push-based architectural style for AJAX applications. J. Syst. Softw. **81**(12), 2194–2209 (2008)
14. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: 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, pp. 2–11. ACM (2010)
15. Realty_rest. https://github.com/ccoenraets/ionic2-realty-rest
16. Search_leak. https://github.com/newarmy/test
17. Selakovic, M., Pradel, M.: Performance issues and optimizations in JavaScript: an empirical study. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 61–72 (2016)
18. Sung, C., Kusano, M., Sinha, N., Wang, C.: Static DOM event dependency analysis for testing web applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 447–459. FSE 2016 (2016)
19. theBrownNode. https://github.com/clintcparker/theBrownNode
20. Vilk, J., Berger, E.D.: Bleak: automatically debugging memory leaks in web applications. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 15–29. ACM (2018)
21. Xiao, X., Han, S., Zhang, C., Zhang, D.: Uncovering JavaScript performance code smells relevant to type mutations. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 335–355. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_18
22. Zhang, H.: On the distribution of software faults. IEEE Trans. Softw. Eng. **34**(2), 301–302 (2008)