Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript

Keheliya Gallaba
University of British Columbia
Vancouver, BC, Canada
kgallaba@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

Ivan Beschastnikh
University of British Columbia
Vancouver, BC, Canada
bestchai@cs.ubc.ca

***Abstract*—JavaScript is a popular language for developing web applications and is increasingly used for both client-side and server-side application logic. The JavaScript runtime is inherently event-driven and callbacks are a key language feature. Unfortunately, callbacks induce a non-linear control flow and can be deferred to execute asynchronously, declared anonymously, and may be nested to arbitrary levels. All of these features make callbacks difficult to understand and maintain.**

We perform an empirical study to characterize JavaScript callback usage across a representative corpus of 138 JavaScript programs, with over 5 million lines of JavaScript code. We find that on average, every 10th function definition takes a callback argument, and that over 43% of all callback-accepting function callsites are anonymous. Furthermore, the majority of callbacks are nested, more than half of all callbacks are asynchronous, and asynchronous callbacks, on average, appear more frequently in client-side code (72%) than server-side (55%). We also study three well-known solutions designed to help with the complexities associated with callbacks, including the error-first callback convention, Async.js library, and Promises. Our results inform the design of future JavaScript analysis and code comprehension tools.

***Index Terms*—Empirical Study, JavaScript, Callback, Asynchrony, Event-driven Programming**

I. Introduction

Callbacks, or higher-order functions, are available in many programming languages, for instance, as function-valued arguments (e.g., Python), function pointers (e.g., C++), and lambda expressions (e.g., Lisp). In this paper we study JavaScript callbacks since JavaScript is the dominant language for building web applications. For example, a recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [31]. The callback language feature in JavaScript is an important factor to its success. For instance, JavaScript callbacks are used to responsively handle events on the client-side by executing functions asynchronously. And, in Node.js[1], a popular JavaScript-based framework, callbacks are used on the server-side to service multiple concurrent client requests.

Although callbacks are a key JavaScript feature, they have not received much attention in the research community. We think that this is a critical omission as the usage of callbacks is an important factor in developer comprehension and maintenance of JavaScript code.

[1] https://nodejs.org

```
1  var db = require('somedatabaseprovider');
2  http.get('/recentposts', function(req, res){
3   db.openConnection('host', creds, function(err, conn){
4    res.param['posts'].forEach(function(post) {
5     conn.query('select * from users where id=' + post
           ['user'], function(err, results){
6           conn.close();
7           res.send(results[0]);
8     });
9    });
10  });
11 });
```

Listing 1. A representative JavaScript snippet illustrating the comprehension and maintainability challenges associated with nested, anonymous callbacks and asynchronous callback scheduling.

Listing 1 illustrates three common challenges with callbacks. First, the three callbacks in this example (on lines 2, 3, and 5) are *anonymous*. This makes the callbacks difficult to reuse and to understand as they are not descriptive. Second, the callbacks in the example are *nested* to three levels, making it challenging to reason about the flow of control in the code. Finally, in line 5 there is a call to `conn.query`, which invokes the second callback parameter *asynchronously*. That is, the execution of the inner-most anonymous function (lines 6–7) is deferred until some later time. As a result, most of the complexity in this small example rests on extensive use of callbacks; in particular, the use of anonymous, nested, and asynchronous callbacks.

Though the issues outlined above have not been studied in detail, they are well-known to developers. Searching for "callback hell" on the web brings up many articles with best practices on callback usage in JavaScript. For example, a prominent problem with asynchronous callbacks in JavaScript is error-handling. The built-in try/catch mechanism does not work with asynchronous callbacks. In response, the JavaScript developer community has, over time, arrived at a best practice known as "error-first protocol". The intent is to reserve the first argument in a callback for communicating errors. However, as this is a best practice and adhering to this protocol is optional, it is not clear to what extent developers use it in practice.

In this paper, we report on an empirical study to characterize JavaScript callback usage across 138 large JavaScript projects. These include 86 Node.js modules from the NPM public package registry used in server-side code and 62 subject systems from a broad spectrum of categories, such as JavaScript

978-1-4673-7899-4/15/$31.00 ©2015 IEEE

MVC frameworks, games, and data visualization libraries. Analyzing JavaScript code statically to identify callbacks and to characterize their properties for such a study presents a number of challenges. For example, JavaScript is a loosely typed language and its functions are variadic, i.e., they accept a variable number of arguments. We developed new JavaScript analysis techniques, building on prior techniques and tools, to identify callbacks and to measure their various features.

The focus of our study is on gaining an understanding of callback usage in practice. We study questions such as, how often are callbacks used, how deep are callbacks nested, are anonymous callbacks more common than named callbacks, are callbacks used differently on the client-side as compared to the server-side, and so on. Finally we measure the extent to which developers rely on the "error-first protocol" best practice, and the adoption of two recent proposals to mitigate callback-related challenges, the Async.js library [21] and Promises [6], a new JavaScript language feature.

The results of our study show that (1) callbacks are passed as arguments more than twice as often in server-side code than in client-side code, i.e., 24% of all server-side call-sites use a callback in server-side code, while only 9% use a callback in client-side code; (2) anonymous callbacks are used in 42% of all callback-accepting function call-sites; (3) there is extensive callback nesting, namely, most callbacks nest 2 levels, and some nest as deep as 8 levels, and (4) there is an extensive use of asynchrony associated with callbacks — 75% of all client-side callbacks were used in conjunction with built-in asynchronous JavaScript APIs.

These results indicate that existing JavaScript analyses and tools [23], [19] often make simplifying assumptions about JavaScript callbacks that might not be true in practice. For example, some of them ignore anonymous callbacks, asynchrony, and callback nesting altogether. Our work stresses the importance of empirically validating assumptions made in the designs of JavaScript analysis tools.

We believe that our characterization of the real-world use of callbacks in different types of JavaScript programs will be useful to tool builders who employ static and dynamic analyses (e.g., which language corner-cases to analyze). Our results will also make language designers more aware of how developers use callback-related language features in practice.

Next, we overview the key features of JavaScript necessary to understand our work.

II. Callbacks in JavaScript

A callback is a function that is passed as an argument to another function, which is expected to invoke it either immediately or at some point in the future. Callbacks can be seen as a form of the continuation-passing style (CPS) [32], in which control is passed explicitly in the form of a continuation; in this case the callback passed as an argument represents a continuation.

Synchronous and asynchronous callbacks. There are two types of callbacks. A callback passed to a function f can be invoked *synchronously* before f returns, or it can be deferred to execute *asynchronously* some time after f returns.

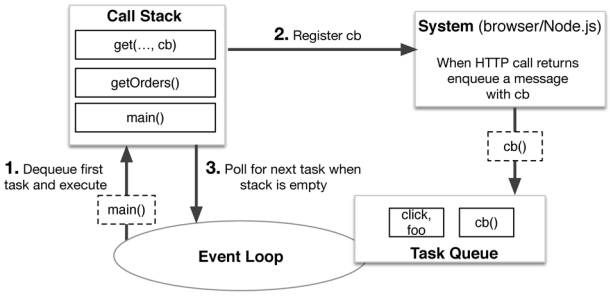

Fig. 1. The JavaScript event loop model.

JavaScript uses an event-driven model with a single thread of execution. Programming with callbacks is especially useful when a caller does not want to wait until the callee completes. To this end, the desired non-blocking operation is scheduled as a callback and the main thread continues its synchronous execution. When the operation completes, a message is enqueued into a task queue along with the provided callback. The event loop in JavaScript prioritizes the single thread to execute the call stack first; when the stack is empty, the event loop dequeues a message from the task queue and executes the corresponding callback function. Figure 1 illustrates the event loop model of JavaScript for a non-blocking HTTP `get` call with a callback named `cb`.

Named and anonymous callbacks. JavaScript callbacks can be *named* functions or *anonymous* functions (e.g., lines 2, 3, or 5 of Listing 1). Each approach has its tradeoffs. Named callbacks can be reused and are easily identified in stack traces or breakpoints during debugging activities. However naming causes the callback function to persist in memory and prevents it from being garbage collected. On the other hand, anonymous callbacks are more resource-friendly because they are marked for garbage collection immediately after being executed. However, anonymous callbacks are not reusable and may be difficult to maintain, test, or debug.

Nested callbacks. Developers often need to combine several callback-accepting functions together to achieve a certain task. For example, two callbacks have to be nested if the result of the first callback needs to be passed into the second callback in a non-blocking way (see lines 2-8 in Listing 3). This structure becomes more complex when the callbacks need to be conditionally nested. Control-flow composition with nested callbacks increases the complexity of the code. The term 'callback hell' [26] has been coined by developers to voice their common frustration with this complexity.

Error-first callbacks. In synchronous JavaScript code the `throw` keyword can be used to signal an error and `try/catch` can be used to handle the error. When there is asynchrony, however, it may not be possible to handle an error in the context it is thrown. Instead, the error must be propagated asynchronously to an error handler in a different context. Callbacks are the basic mechanism for delivering errors asynchronously

in JavaScript. Because there is no explicit language support for asynchronous error signaling, the developer community has proposed a convention — dedicate the first argument in the callback to be a permanent place-holder for error signaling.

More exactly, the *error-first callback* protocol specifies that during a callback invocation, either the first error argument is non-null (indicating an error), or the error argument is null and the other arguments contain data (indicating success), but not both [1]. Listing 2 shows an example of this protocol. If an error occurs while reading the file (Line 4), the anonymous function will be called with error as the first argument. For this error to be handled, it will be propagated by passing it as the first argument of the callback (in line 5). The error can then be handled at a more appropriate location (lines 17–21). When there is no error, the program continues (line 8) and invokes the callback with the first (error) argument set to null.

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Listing 2. Error-first callback protocol.

The error-first callback protocol is intended to simplify exception handling for developers; if there is an error, it will always propagate as the first argument through the API, and API clients can always check the first argument for errors. But, there is no automatic checking of the error-first protocol in JavaScript. It is therefore unclear how frequently developers adhere to this protocol in practice.

Handling callbacks. A quick search in the NPM repository[2] reveals that there are over 4,500 modules to help developers with asynchronous JavaScript programming. Two solutions that are gaining traction in helping developers handle callbacks are libraries, such as *Async.js* [21] and new language features such as *Promises* [6]. We now briefly detail these two approaches.

The Async.js library exposes an API to help developers manage callbacks. For example, Listing 3 shows how nested callbacks in vanilla JavaScript (lines 1–8) can be expressed using the `waterfall` method available in Async.js (11–18).

[2] https://www.npmjs.com/

```
1  // Before: nested callbacks
2  $("#button").click(function() {
3      promptUserForTwitterHandle(function(handle) {
4          twitter.getTweetsFor(handle, function(tweets) {
5              ui.show(tweets);
6          });
7      });
8  });
9
10 // After: Using Async.js waterfall method
11 $("#button").click(function() {
12     async.waterfall([
13         promptUserForTwitterHandle,
14         twitter.getTweetsFor,
15         ui.show
16     ]
17     , handleError);
18 });
19
20 // After: sequential join of callbacks with Promises
21 $("#button").clickPromise()
22     .then(promptUserForTwitterHandle)
23     .then(twitter.getTweetsFor)
24     .then(ui.show);
```

Listing 3. Rewriting nested callbacks using Async.js or Promises.

Promises are a JavaScript language extension. A Promise-based function takes some input and returns a promise object representing the result of an asynchronous operation. A promise object can be queried by the developer to answer questions like "were there any errors while executing the async call?" or "has the data from the async call arrived yet?" A promise object, once fulfilled, can notify any function that depends on its data. Listing 3 illustrates how nested callbacks in vanilla JavaScript (lines 1–8) can be re-expressed using Promises (20–24).

Next, we describe the methodology underlying our study of JavaScript callbacks.

III. Methodology

To characterize callback usage in JavaScript applications, we focus on the following three research questions.

RQ1: How prevalent are callbacks?
RQ2: How are callbacks programmed and used?
RQ3: Do developers use external APIs to handle callbacks?

Our analyses are open source [4] and all of our empirical data is available for download [5].

A. Subject Systems

We study 138 popular open source JavaScript subject systems from six distinct categories: NPM modules (86), web applications (16), game engines (16), client-side frameworks (8), visualization libraries (6), and games (6). NPM modules are used only on the server-side. Client-side frameworks, visualization libraries, and games include only client-side code. Web applications and game engines include both client-side and server-side code. Table I presents these categories, whether or not the category includes client-side and server-side application code, and the aggregate number of JavaScript files and lines of JavaScript code that we *analyze* for each applications category.

TABLE I
JAVASCRIPT SUBJECT SYSTEMS IN OUR STUDY

Category	Subject systems	Client side	Server side	Total files	Total LOC
NPM Modules	86		✓	4,983	1,228,271
Web Apps.	16	✓	✓	1,779	494,621
Game Engines	16	✓	✓	1,740	1,726,122
Frameworks	8	✓		2,374	711,172
DataViz Libs.	6	✓		3,731	958,983
Games	6	✓		347	119,279
Total	**138**	✓	✓	**14,954**	**5,238,448**

The 86 NPM modules we study are the most depended-on modules in the NPM repository [3]. The other subject systems were selected from *GitHub Showcases* [2], where popular and trending open source repositories are organized around different topics. The subject systems we consider are JavaScript-only systems. Those systems that contain server-side components are written in Node.js[3], a popular server-side JavaScript framework. Overall, we study callbacks in over 5 million lines of JavaScript code.

B. Analysis

To address the three research questions, we have developed a static analyzer to search for different patterns of callbacks in JavaScript code[4].

Our static analyzer builds on prior JavaScript analysis tools, such as Esprima [15] to parse and build an AST, Estraverse [33] to traverse the AST, and TernJS [14], a type inference technique by Hackett and Guo [13], to query for function type arguments. We also developed a custom set of analyses to identify callbacks and to measure their various properties of interest.

Duplicate code is an issue for any source code analysis tool that measures the prevalence of some language feature. Our analysis maintains a set of dependencies for a subject system and guarantees that each dependency is analyzed exactly once. To resolve dependencies expressed with the `require` keyword, we use TernJS and its *Node* plugin.

In the rest of this section we detail our analysis for each of the three research questions.

Prevalence of callbacks (RQ1). To investigate the prevalence of callbacks in JavaScript code we consider all function definitions and function callsites in each subject system. For each subject, we compute (1) the percentage of function definitions that accept callbacks as arguments, and (2) the percentage of function callsites that accept callbacks as arguments.

We say that f is a *callback-accepting function* if we find that at least one argument to f is used as a callback.

To determine whether a parameter p of a function f definition is a callback argument, we use a three-step process: (1) if p is invoked as a function in the body of f then p is a callback; (2) if p is passed to a known callback-accepting function (e.g., `setTimeout`) then p is a callback; (3) if p is used as an argument to an unknown function f', then recursively check if p is a callback parameter in f'.

Note that for f to be a callback-accepting function, it is insufficient to find an invocation of f with a function argument p. To be a callback-accepting function, the argument p must be invoked inside f, or p must be passed down to some other function where it is invoked.

We do not analyze a subject's dependencies, such as libraries, to find callbacks[5]. The only time we analyze external libraries is when a subject system calls into a library and passes a function as an argument. In this case our analysis check whether the passed function is invoked as a callback in the library code.

We also study whether callback usage patterns are different between server-side and client-side JavaScript code. Categorizing the projects known as purely client-side (MVC frameworks like Ember, Angular) or purely server-side (NPM modules) is easy. But, some projects contain both server-side and client-side JavaScript code. To distinguish client-side code from server-side code, we use the project directory structure. We assume that client-side code is stored in a directory named *www*, *public*, *static*, or *client*. We also identify client-side code through developer code annotations (e.g., `/* jshint browser:true, jquery:true */`).

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

Listing 4. An example of an asynchronous anonymous callback.

For example, consider the code in Listing 4. To check if the `getRecord()` function invocation in line 11 takes a callback, we analyze its definition in line 1. We find that there is a path from the start of `getRecord()` to the `callback` invocation (which happens to be the `logStuff` argument provided to the `getRecord()` invocation). The path is: $getRecord() \rightarrow http.get() \rightarrow Anonymous1() \rightarrow callback()$. Therefore, `logStuff()` is a callback function because it is passed as a function argument and it is invoked in that function. This makes `getRecord()` a callback-accepting function.

We label a *callsite* as callback-accepting if it corresponds to (1) a function that we determine to be callback-accepting, as described above, or (2) a function known a-priori to be callback-accepting (e.g., `setTimeout()`). The `getRecord` callsite in line 11 is callback-accepting because the function `getRecord` was determined to be callback-accepting.

Callback usage in practice (RQ2). Developers use callbacks in different ways. Some of these are known to be problematic [26] for comprehension and maintenance (e.g.,

[3] https://nodejs.org

[4] Our static analysis approach considers most program paths, but it does not handle cases like `eval` that require dynamic analysis.

[5] For instance, JavaScript files under the *node_modules* directory are excluded.

see section II). We characterize callback usage in three ways: we compute (1) the percentage of callbacks that are anonymous versus named, (2) the percentage of callbacks that are asynchronous, and (3) the callback nesting depth.

Anonymous versus named callbacks. If a function callsite is identified as callback-accepting, and an anonymous function expression is used as an argument, we call it an instance of an anonymous callback.

Asynchronous callbacks. A callback passed into a function can be deferred and invoked at a later time. This deferral happens through known system APIs, which deal with the task queue in common browsers and Node.js environments. Our analysis detects a variety of APIs, including DOM events, network calls, timers, and I/O. Table II lists examples of these asynchronous APIs.

TABLE II
EXAMPLE ASYNCHRONOUS APIS AVAILABLE TO JAVASCRIPT PROGRAMS

Category	Examples	Availability
DOM events	`addEventListener`, `onclick`	Browser
Network calls	`XMLHTTPRequest.open`	Browser
Timers (macro-Task)	`setImmediate()`, `setTimeout()`, `setInterval()`	Browser, Node.js
Timers (micro-task)	`process.nextTick()`	Node.js
I/O	APIs of `fs`, `net`	Node.js

For each function definition, if a callback argument is passed into a known deferring function call, we label this callback as asynchronous.

Callback nesting. The number of callbacks nested inside one after the other is defined as the callback depth. According to this definition, Listing 1 has a callback depth of three because of callbacks at lines 2, 3, and 5.

In cases where there are multiple instances of nested callbacks in a function, we count the maximum depth of nesting, including nesting in conditional branches. This under-approximates callback nesting. For example, Listing 5 shows an example code snippet from an open source application[6]. There are two sets of nested callbacks in this example, namely, at Lines 1, 4, 5 (depth of three) and a second set at Lines 1, 12, 13, 15 (depth of four). Our analysis computes the maximum nesting depth for this function, which is four.

Handling callbacks (RQ3). There are a number of solutions to help with the complexities associated with callbacks. We consider three well-known solutions: (1) the prevalence of the error-first callback convention, (2) the prevalence and usage of Async.js [21], a popular control flow library, and (3) the prevalence of Promises [6], a recent language extension, which provides an alternative to using callbacks. For each solution, we characterize its usage — are developers using the solution and to what extent.

[6]https://github.com/NodeBB/NodeBB

```
1  define('admin/general/dashboard', 'semver', function(semver
       ) {
2    var Admin = {};

4    $('#logout-link').on('click', function() {
5      $.post(RELATIVE_PATH + '/logout', function() {
6        window.location.href = RELATIVE_PATH + '/';
7      });
8    });

10   ...

12   $('.restart').on('click', function() {
13     bootbox.confirm('Are you sure you wish to restart
           NodeBB?', function(confirm) {
14       if (confirm) {
15         $(window).one('action:reconnected', function() {
16           app.alert({ alert_id: 'instance_restart', });
17         });

19         socket.emit('admin.restart');
20       }
21     });
22   });
23   return Admin;
24 });
```

Listing 5. Example of multiple nested callbacks in one function.

Error-first callbacks. To detect error-first callbacks (see section II) we use a heuristic. We check if the first parameter p of a function f definition has the name '*error*' or '*err*'. Then, we check if f's callsites also contain '*error*' or '*err*' as their first argument. Thus, our analysis counts the percentage of function definitions that accept an error as the first argument, as well as the percentage of function callsites that are invoked with an error passed as the first argument.

Async.js. If a subject system has a dependency on Async.js (e.g., in their `package.json`), we count all invocations of the Async.js APIs in the subject's code.

Promises. We count instances of Promise creation and consumption for each subject system. If a new expression returns a Promise object (e.g., `new Promise()`), it is counted as a Promise creation. Invocations of the `then()` method on a Promise object are counted as Promise consumption — this method is used to attach callbacks to a promise. These callbacks are then invoked when a promise changes its state (i.e., evaluates to success or an error).

IV. RESULTS

A. Prevalence of Callbacks (RQ1)

In the subject systems that we studied, on average, 10% of all function definitions and 19% of all function callsites were callback-accepting. Figure 2 depicts the percentage of callback-accepting function definitions and callsites, per category of systems (Table I), and in aggregate. The figure also shows how these are distributed across client-side and server-side, indicating that server-side code generally contains more functions that take callbacks than client-side code.

***Finding 1**: On average, every 10th function definition takes a callback argument. Callback-accepting function* definitions *are more prevalent in server-side code (10%) than in client-side code (4.5%).*

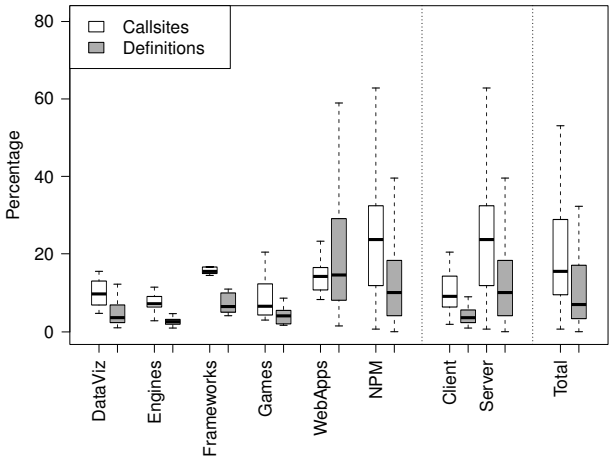

Fig. 2. Boxplots for percentage of callback-accepting function definitions and callsites per category, across client/server, and in total.

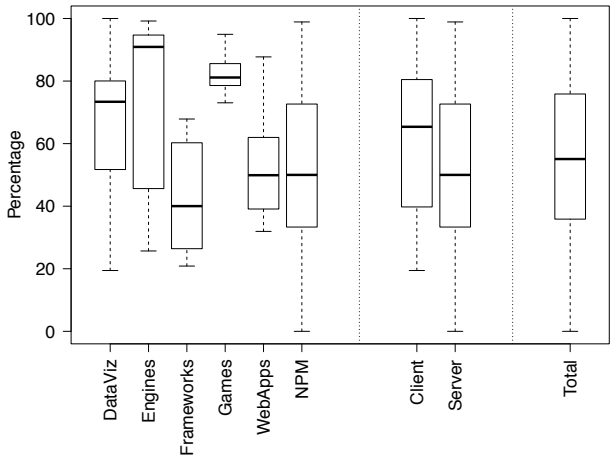

Fig. 3. Boxplots for percentage of asynchronous callback-accepting function callsites.

Finding 2*: On average, every 5th function callsite takes a callback argument. Callback-accepting function* callsites *are more prevalent in server-side code (24%) than in client-side code (9%).*

Implications. Callbacks are utilized across all subject categories we considered. Some categories contain a higher degree of callback usage. For example, the web applications category contained a higher fraction of both callback-accepting function definitions and invocations of such functions. The inter-category differences in callback usage were not large, however, and we believe that these differences can be ascribed to the fact that some categories contain more server-side code than others. We believe the more extensive usage of callbacks in server-side code can be attributed to the continuation-passing-style of programming that was advocated in the Node.js community from its inception.

B. Callback Usage — Asynchronous Callbacks (RQ2)

As a reminder, an asynchronous callback is a callback that is eventually passed to an asynchronous API call like `setTimeout()` (see subsection III-B for more details).

Figure 3 shows the prevalence of asynchronous callback accepting function callsites. Across all applications there was a median of 56% and a mean of 56% of callsites with asynchronous callbacks. The figure also partitions the data into the client-side and server-side categories. We find that usage of asynchronous callbacks is higher in client-side code. Of all callsites in client-side code, 72% were asynchronous. On the server-side, asynchronous callbacks usage is 55% on average.

Finding 3*: More than half of all callbacks are asynchronous. Asynchronous callbacks, on average, appear more frequently in client-side code (72%) than server-side code (55%).*

Implications. The extensive use of asynchrony in the subjects we studied indicates that program analyses techniques that ignore the presence of asynchrony are inapplicable and may lead to poor results. Analyses of JavaScript must account for asynchrony.

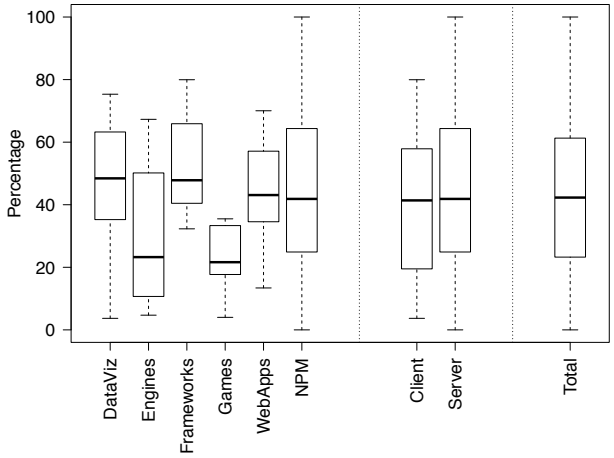

Fig. 4. Boxplots for percentage of anonymous callback-accepting function callsites per category, across client/server, and in total.

The extensive use of asynchronous scheduling paired with callbacks surprised us. The asynchronous programming style significantly complicates program control flow and impedes program comprehension [8]. Yet there are few tools to help with this. We think that the problem of helping developers reason about large JavaScript code bases containing asynchronous callbacks, both on the client-side and the server-side, deserves more attention from the research community.

C. Callback Usage — Anonymous Callbacks (RQ2)

Figure 4 shows the prevalence of anonymous callback-accepting function callsites. The median percentage across the categories ranges from 23% to 48%, which is fairly high considering that anonymous callbacks are difficult to understand and maintain. Figure 4 also shows the same data partitioned between client-side and server-side code, and indicates that server-side code contains a slightly higher percentage of anonymous callbacks than client-side code.

***Finding 4**: Over 43% of all callback-accepting function callsites are invoked with at least one anonymous callback. There is little difference between client-side and server-side code in the extent to which they use anonymous callbacks.*

Implications. This finding indicates that in a large fraction of cases, a callback is used once (anonymously) and is never reused again. It seems that developers find anonymous callbacks useful in spite of the associated comprehension, debugging, and testing challenges. We think that this topic deserves further study — it is important to understand why developers use anonymous callbacks and prefer them over named callbacks. Possible reasons for using anonymous callbacks could be code brevity, or creating temporary local scopes (e.g., in closures). We also think that the high fraction of anonymous callbacks indicates that this popular language feature is here to stay. Therefore, it is worthwhile for the research community to invest time in developing tools that will support developers in handling anonymous callbacks.

D. Callback Usage — Nested Callbacks (RQ2)

Figure 5 presents our results for the total number of instances of nested callbacks at each observed nesting level. We found that the majority of callbacks nest two levels deep. Figure 5 shows this unusual peak at nesting level of 2. We also found that callbacks are nested up to a depth of 8 (there were 29 instances of nesting at this level). In these extreme cases developers compose sequences of asynchronous callbacks with result values that flow from one callback into the next. These extreme nesting examples are available as part of our dataset [5].

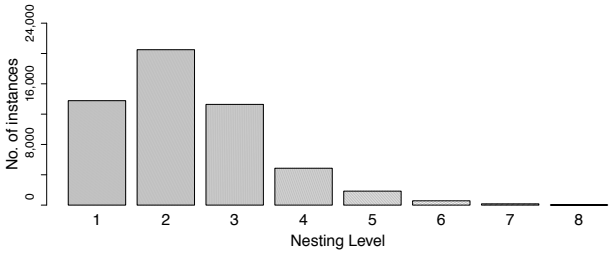

Fig. 5. Instances of nested callbacks for a particular nesting level.

***Finding 5**: Callbacks are nested up to a depth of 8. There is a peak at nesting level of 2.*

Implications. As with anonymous and asynchronous callbacks, callback nesting taxes code readability and comprehension. We find that nesting is widely used in practice and note that developers lack tools to manage callback nesting. We believe that there is ample opportunity in this area for tool builders and software analysis experts. The number of instances decreases from level 1 to 8, except at level 2. Based on our investigation of numerous level 2 callback nesting examples, we believe that the peak at level 2 is due to a common JavaScript practice in which project code is surrounded with an anonymous function from an external library. This is used, for example, for module exporting, loading, or to wait for the DOM to be fully loaded on the client-side. Due to the extra callback surrounding the project code, in these type of projects, callbacks begin nesting at level 2. Listing 6 lists an example of this kind of nesting with the `$(document).ready()` function (line 1) from the popular jQuery library. This function waits for the DOM to load. It increases the callback nesting in the rest of the code by 1 (e.g., the callback on line 2 has a nesting level of 2).

```
1 $(document).ready(function () {
2   $('.star').click(function (e) {
3     ...
4   })
5 })
```

Listing 6. Example of a nested callback introduced by the wrapping `$(document).ready()` function from the jQuery library.

E. Solutions — Error-first Protocol (RQ3)

We found that 20% of all function definitions follow the error-first protocol. The median percentage across the categories ranges from 4% to 50%. The fraction of function definitions that adhere to the error-first protocol is almost twice as high in the server-side code (30%) than in the client-side code (16%). In addition, the error-first protocol was the most common solution among the three solutions we considered. For example, 73% (63 out of 86) NPM modules and 93% (15 out of 16) web applications had instances of the error-first protocol.

***Finding 6**: Overall, every 5^{th} function definition adheres to the error-first protocol. The error-first protocol is used twice as often in server-side code than in client-side code*

Implications. Although we found that a non-trivial fraction of JavaScript functions rely on the error-first protocol, it remains an ad-hoc solution that is loosely applied. The relatively low and highly variable use of the error-first protocol means that developers must check adherence manually and cannot depend on APIs and libraries to enforce it. Such idiom-based strategies for handling exceptions are known to be error-prone in other languages, such as C [10]. It would be interesting to study if this is also the case for JavaScript functions that follow the error-first callback idiom.

F. Solutions — Async.js (RQ3)

To study Async.js, we considered subject systems that use this library. We found that only systems in the web applications and NPM modules categories used this library. Our results show that 9 of 16 (56%) web applications and just 9 of 85 (11%) NPM modules use the Async.js library to manage asynchronous control flow. Table III shows the top 10 used functions from the Async.js API (by number of callsites) for these two categories of subject systems.

This table indicates that the sets of functions used in the top 10 list are similar. But, there are notable differences: for example, `nextTick` was the most used Async.js method in

TABLE III
TOP 10 ASYNC.JS INVOKED METHODS IN JAVASCRIPT WEB APPLICATIONS (LEFT) AND NPM MODULES (RIGHT). THE * SYMBOL DENOTES CALLS THAT DO NOT APPEAR IN BOTH TABLES.

Rank	Method	Count	Rank	Method	Count
1	nextTick	18	1	parallel	189
2	queue*	16	2	apply	81
3	each	14	3	waterfall	72
3	setImmediate*	14	4	series	61
3	series	14	5	each	48
6	auto*	11	6	map	37
6	waterfall	11	7	eachSeries*	20
6	parallel	11	8	eachLimit*	12
9	map	10	9	whilst*	10
9	apply	10	9	nextTick	10

web applications and just the 9^{th} most used method in NPM modules. The `nextTick` method in Async.js is used to delay the invocation of the callback until a later tick of the event loop, which allows other events to precede the execution of the callback. In Node.js code the `nextTick` is implemented using the `process.nextTick()` method in the runtime. In browsers this call is implemented using `setImmediate(callback)` or `setTimeout(callback, 0)`. In this case Async.js provides a single interface for developers to achieve the same functionality in both client-side and server-side code.

In NPM modules `parallel` is the most widely used Async.js method (it is the 6^{th} most popular among web applications). This call is used to run an array of independent functions in parallel.

As a final example, the second-most used call in web-applications, `queue`, does not appear in the top ten calls used by NPM modules. The `queue` method functionality is similar to that of `parallel`, except that tasks can be added to the queue at a later time and the progress of tasks in the queue can be monitored.

We should note that because JavaScript is single-threaded, both the `parallel` and `queue` Async.js calls do not expose true *parallelism*. Here, parallel execution means that there may be points in time when two or more tasks have started, but have not yet completed.

There are significant differences between web applications and NPM modules in terms of the Async.js API usage. To characterize this difference, we first ranked the API functions according to the number of times they were used in web applications and NPM modules. Then we analyzed the difference between the ranks of each function in the two categories. For example, the rank of `nextTick` is 9 and 1 in the NPM modules and web applications, respectively, making the absolute difference 8. Overall, the rank differences had a mean of 6.2, a median of 5.5, and a variance of 23.8. This indicates that the Async.js library is used differently in these two categories of subject systems.

***Finding 7**: More than half of the web applications (56%) use the Async.js library to manage asynchronous control flow. The usage is much lower (11%) in the NPM modules. In addition, the Async.js library is used differently (rank variance of 23.8) in these two categories of subject systems.*

Implications. Libraries, such as Async.js, provide one means of coping with the complexity of callbacks. However, because library solutions are not provided natively by the language runtime, the developer community can be divided on which library to use, especially as there are many alternatives (see section II). We think that the difference in Async.js API usage by developers of web applications (that include both client-side code and server-side code) and NPM modules (exclusively server-side code) indicates different underlying concerns around callbacks and their management. We think that this usage deserves further study and can inform the design of future libraries and proposals for language extensions [18], [22], [7].

G. Solutions — Promises (RQ3)

Table IV shows the percentage of subjects that create promises and the percentage of subjects that use promises.

TABLE IV
PERCENTAGE OF SUBJECT SYSTEMS CREATING AND USING PROMISES

Category	Subjects creating Promises (%)	Subjects using Promises (%)
DataViz libraries	6	31
Game Engines	0	25
Frameworks	50	75
Games	0	17
Web Applications	13	50
NPM Modules	3	12
Total	8	26

Figure 6 shows box plots for the number of Promise creating instances using `new Promise()` constructs and Promise usage, e.g., `then()`, in the different subject categories, partitioned across client/server, and in total. It should be noted that not all application access a Promise through the `new Promise()` statement; some invoke library functions that return Promises.

In aggregate, we found that 37 of 138 (27%) applications use Promises. They were predominantly used by client-side frameworks (75%), with a maximum of 513 usage instances (across all frameworks) and a standard deviation of 343 usage instances. In all the other subject systems, usage of Promises was rare, with a mean close to zero. There was one outlier, the bluebird NPM module[7], that had 2,032 Promise usage instances. This module implements the Promises specification as a library. We therefore omit it from our results.

***Finding 8**: 27% of subject systems use Promises. This usage is concentrated in client-side code, particularly in JavaScript frameworks.*

[7] https://www.npmjs.com/package/bluebird

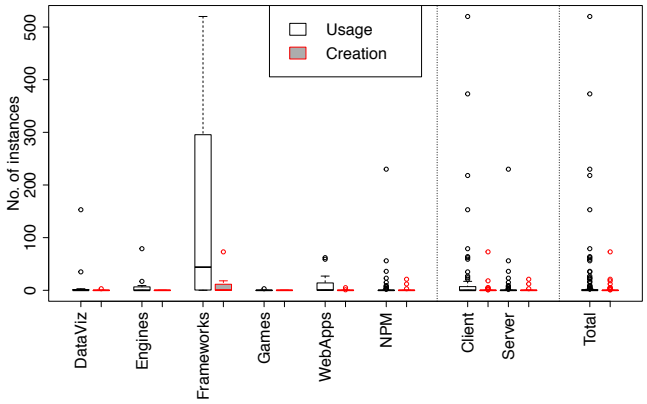

Fig. 6. The distribution of total Promise usage and creation instances by category, across client/server, and in total.

Implications. Although Promises is a promising language-based approach to resolving many of the challenges related to callbacks, such as nesting and error handling, we have not observed a significant uptake of Promises among the systems we studied. This could be because Promises is a relatively new addition to browsers and Node.js. It would be interesting to study how this adoption evolves and whether Promises lead to higher quality and more maintainable JavaScript code. Tools that automatically refactor callbacks into Promises would help developers to migrate existing large projects to use Promises.

V. Threats to validity

There are both internal and external threats to validity for our study. We overview these threats in this section.

Internal threats. Our analyses rely on some development conventions. The error-first callback analysis uses a naming heuristic: code adhering to the error-first protocol will name the first argument of a function as *err* or *error*. A threat is that we may be under-counting protocol adherence by missing cases where the protocol is followed but a different argument name is used. And, we may also be over-counting adherence, since an argument name does not necessarily mean that the code uses the protocol, or that it properly follows it. We also rely on directory naming conventions and use code annotations as hints to identify client-side and server-side code.

We decided to count features of callback usage in particular ways. For example, we count callback nesting by taking the maximum depth of callback nesting for a function. This can provide an under approximation of the number of instances of nested callbacks.

Our analyses are static. This limits the kinds of JavaScript behaviors that we can analyze. For example, we do not handle code in `eval` statements in our study.

External threats. Although we study over 5 million lines of JavaScript code, our sample might not be representative, in a number of ways. First, it comes from open source projects of a particular size and maturity. Second, we consider projects that use JavaScript and are primarily mono-lingual. For example, we do not consider projects that use JavaScript on the client-side and Java on the server-side. As a result, our findings may not generalize to other types of JavaScript projects. However, the subject systems in our study represent five different categories and as the first study of its kind, we believe our characterization study of JavaScript callback usage in practice is worthwhile, and hope that it will lead to other studies that consider a broader variety of subject systems.

All our empirical data and toolset are publicly available; since the subject systems are all open source, our study should be repeatable.

VI. Related Work

Researchers have studied various aspects of the JavaScript language in real-world applications and callback-related issues are a recurrent discussion topic among developers [26]. However, to the best of our knowledge, there have been no empirical studies of callback usage in practice.

JavaScript applications. The dynamic behaviour of JavaScript applications was studied by Richards et al. [30]. They found that commonly made assumptions about dynamism in JavaScript are violated in at least some real-world code. A similar study was conducted by Martinsen et al. [20]. Richards et al. [29] studied the prevalence of `eval`. They found `eval` to be pervasive, and argued that in most usage scenarios, it could be replaced with equivalent and safer code or language extensions.

Ocariza et al. [25] conducted an empirical study to characterize root causes of client-side JavaScript bugs. Since this study, server-side JavaScript, on top of Node.js, has gained traction among developers. Our study considers callback usage in both client- and server-side JavaScript code.

Security vulnerabilities in JavaScript code have also been studied. Examples include studies on remote JavaScript inclusions [24], [35], cross-site scripting (XSS) [34], and privacy-violating information flows [17]. Parallelism in JavaScript code was studied by Fortuna et al. [12].

Milani Fard et al. [23] studied code smells in JavaScript code. In their list of JavaScript smells, they included nested callbacks, but only focus on callbacks in client-side code. Decofun [11] is a JavaScript function de-anonymizer. It parses the code and names any detected anonymous function according to its context. Brodu et al. [9] propose a compiler for converting nested callbacks, or an imbrication of continuations, into a sequence of Dues, which is a simpler version of Promises.

Asynchronous programming. Okur et al. [27] recently conducted a large-scale study on the usage of asynchronous programming in C# applications. They found that callback-based asynchronous idioms are heavily used, but new idioms that can take advantage of the `async/await` keywords are rarely used in practice. They have studied how developers (mis)use some of these new language constructs. Our study similarly covers the usage of callbacks and new language features such as Promises to enhance asynchronous programming. However, our work considers JavaScript code and delves deeper. For example, we characterize the usage of callback nesting and

anonymous callbacks, which are known to cause maintenance problems.

Concurrency bug detection. EventRacer [28] detects data races in JavaScript applications. Zheng et al. [36] propose a static analysis method for detecting concurrency bugs caused by asynchronous calls in web applications. Similarly, WAVE [16] is a tool for identifying concurrency bugs by looking for the same sequence of user events leading to different final DOM-trees of the application.

Program comprehension. Clematis [8] is a technique for helping developers understand complex event-based and asynchronous interactions in JavaScript code by capturing low-level interactions and presenting those as higher-level behavioral models. Theseus [19] is an IDE extension that helps developers to navigate asynchronous and dynamic JavaScript execution. Theseus has some limitations; for example, it does not support named callbacks. Our study demonstrates that over half of all callbacks are named, indicating that many applications will be negatively impacted by similar limitations in existing tools.

VII. Conclusions

All modern JavaScript applications that handle and respond to events use callbacks. However, developers are frequently frustrated by "callback hell" — the comprehension and maintainability challenges associated with nested, anonymous callbacks and asynchronous callback scheduling. This paper presents an empirical study of callbacks usage in practice. We study over 5 million lines of JavaScript code in 138 subject systems that span a variety of categories. We report on the prevalence of callbacks, their usage, and the prevalence of solutions that help to manage the complexity associated with callbacks. We hope that our study will inform the design of future JavaScript analysis and code comprehension tools. Our analysis [4] and empirical results [5] are available online.

Acknowledgments

This work was supported in part by an NSERC Strategic Grant and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

[1] Error Handling in Node.js. https://www.joyent.com/developers/node/design/errors, 2014.
[2] Github Showcases. https://github.com/showcases, 2014.
[3] Most depended-upon NMP packages. https://www.npmjs.com/browse/depended, 2014.
[4] CallMeBack. https://github.com/saltlab/callmeback, 2015.
[5] Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. Dataset release. http://salt.ece.ubc.ca/callback-study/, 2015.
[6] Promises/A+ Promise Specification. https://promisesaplus.com, 2015.
[7] TypeScript. http://www.typescriptlang.org, 2015.
[8] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript Event-based Interactions. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.
[9] E. Brodu, S. Frénot, and F. Oblé. Toward Automatic Update from Callbacks to Promises. In *Proc. of the Workshop on All-Web Real-Time Systems (AWeS)*, pages 1:1–1:8, New York, NY, USA, 2015. ACM.
[10] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering Faults in Idiom-based Exception Handling. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 242–251. ACM, 2006.
[11] D. M. Clements. Decofun. https://github.com/davidmarkclements/decofun.
[12] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In *Proc. of Intl. Symposium on Workload Characterization (IISWC)*, pages 1–10, 2010.
[13] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012.
[14] M. Haverbeke. Tern. https://github.com/marijnh/tern, 2015.
[15] A. Hidayat. Esprima. https://github.com/jquery/esprima, 2015.
[16] S. Hong, Y. Park, and M. Kim. Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *Proc. of Intl. Conf. on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, 2014.
[17] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. of Conf. on Comp. and Communications Security*, pages 270–283. ACM, 2010.
[18] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing JavaScript with arrows. *ACM Sigplan Notices*, 44(12):49–58, 2009.
[19] T. Lieber. Theseus: understanding asynchronous code. In *Proc. of the Conf. on Human Factors in Computing Systems*, pages 2731–2736. ACM, 2013.
[20] J. Martinsen, H. Grahn, and A. Isberg. A comparative evaluation of JavaScript execution behavior. In *Proc. of Intl. Conf. on Web Engineering (ICWE)*, pages 399–402. Springer, 2011.
[21] C. McMahon. Async.js. https://github.com/caolan/async.
[22] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proc. of the Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–20, New York, NY, USA, 2009. ACM.
[23] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript Code Smells. In *Proc. of the Intl. Conf. on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE, 2013.
[24] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. of the Conf. on Computer and Comm. Security*, pages 736–747. ACM, 2012.
[25] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *Proc. of the ACM/IEEE Intl. Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE, 2013.
[26] M. Ogden. Callback Hell. http://callbackhell.com, 2015.
[27] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A Study and Toolkit for Asynchronous Programming in C#. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 1117–1127. ACM, 2014.
[28] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proc. of the Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 151–166. ACM, 2013.
[29] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, pages 52–78. Springer, 2011.
[30] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010.
[31] Stack Overflow. Developer Survey. http://stackoverflow.com/research/developer-survey-2015, 2015.
[32] G. J. Sussman and G. L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
[33] Y. Suzuki. Estraverse. https://github.com/estools/estraverse, 2015.
[34] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical Report EECS-2011-11, UC Berkeley, 2011.
[35] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proc. of Intl. Conf. on World Wide Web (WWW)*, pages 961–970. ACM, 2009.
[36] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. of the Intl. Conf. on World Wide Web (WWW)*, pages 805–814. ACM, 2011.