Node.fz: Fuzzing the Server-Side Event-Driven Architecture

James Davis
Virginia Tech
davisjam@vt.edu

Arun Thekumparampil *
MathWorks
arunkt@vt.edu

Dongyoon Lee
Virginia Tech
dongyoon@vt.edu

Abstract

The importance of the Event-Driven Architecture (EDA) has never been greater. Web servers and the IoT alike have begun to adopt the EDA, and the popular server-side EDA framework, Node.js, boasts the world's largest package ecosystem. While multi-threaded programming has been well studied in the literature, concurrency bug characteristics and useful development tools remain largely unexplored for server-side EDA-based applications.

We present the first (to the best of our knowledge) concurrency bug characteristic study of real world open-source event-driven applications, based in Node.js. Like multi-threaded programs, event-driven programs are prone to concurrency bugs like atomicity violations and order violations. Our study shows the forms that atomicity violations and ordering violations take in the EDA context, and points out the limitations of existing concurrency error detection tools developed for client-side EDA applications.

Based on our bug study, we propose *Node.fz*, a novel testing aid for server-side event-driven programs. *Node.fz* is a schedule fuzzing test tool for event-driven programs, embodied for server-side Node.js programs. *Node.fz* randomly perturbs the execution of a Node.js program, allowing Node.js developers to explore a variety of possible schedules. Thanks to its low overhead, *Node.fz* enables a developer to explore a broader "schedule space" with the same test time budget, ensuring that applications will be stable in a wide variety of deployment conditions. We show that *Node.fz* can expose known bugs much more frequently than vanilla Node.js, and that it can uncover new bugs.

* Work done while author was at Virginia Tech.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064188

1. Introduction

The Event-Driven Architecture (EDA) has escaped from the client-side. While traditionally used to build user interfaces in areas like the desktop [50], mobile [10, 33] and web [21, 23], the EDA is now being widely adopted to build general applications like web servers and Internet of Things (IoT) applications. The use of the EDA on the server-side had been promoted through the wide-spread use of the Node.js framework [8]. The Node.js package ecosystem, npm, is the largest ever, with over 400,000 packages [7] and over 1.75 billion package downloads per week[1]. Node.js has been deployed in industry, including at eBay [40], PayPal [22], and LinkedIn [37], and is also being embraced on IoT platforms including Cylon.js [1] and IBM's Node-Red [3].

Event-driven programs, like multi-threaded programs, can have concurrency bugs like atomicity violations and ordering violations [24, 29]. Just as thread-based programs can have race conditions between *unordered threads*, event-driven programs may have them between *unordered events*. The resulting concurrency errors have serious consequences, including server crashes and inconsistent database states, which this paper demonstrates with real examples in §3. Though techniques for detecting concurrency errors in event-driven client-side web [27, 43, 44] and mobile [11, 25, 31] applications have been proposed, server-side event-driven programs have hitherto remained unexplored.

In §3 we study concurrency bug patterns, bug manifestations, and fix strategies in real world open-source npm modules and Node.js programs. Our findings reveal the form that atomicity and ordering violations take in the EDA setting. In addition, we identify three significant differences between client-side event-driven applications and server-side Node.js applications, limiting the applicability of existing bug detection and analysis techniques. First, server programs interact frequently with other system components like databases and the file system. Thus, Node.js programs are an "open system", making existing model checking [27] techniques difficult to apply. Second, we demonstrate that race conditions in Node.js programs are not only on shared memory (e.g. writes to variables and arrays), but also on system resources (e.g. queries to a database, I/O to the file system). Exist-

[1] See https://www.npmjs.com/.

ing related data race detectors (e.g. [43, 44]) consider only memory accesses and would therefore miss many race conditions. Third, server-side programs are much longer-lived than client-side programs, with normal lifetimes spanning thousands or millions of events. Existing approaches all suffer from scalability issues, making them infeasible in the Node.js setting.

To address these issues, we present *Node.fz* (§4), a novel schedule fuzzing test tool for server-side event-driven Node.js applications. *Node.fz* perturbs the execution of a Node.js program, allowing Node.js developers to explore a variety of possible schedules. *Node.fz* thus enables a developer to explore a broader "schedule space", ensuring that an application will be stable on a wide variety of deployment conditions. Our results show that *Node.fz* can expose known bugs more frequently than Node.js, and that it can expose new bugs in two popular npm modules. Critical to easy adoption, *Node.fz* is a drop-in replacement for Node.js and offers comparable performance. In summary, this paper makes the following contributions:

- We present the first concurrency bug characteristic study of real world open-source Node.js programs, illustrating the forms that atomicity violations and ordering violations take in the EDA setting.
- We present *Node.fz*, the first concurrency fuzz testing tool tailored to server-side event-driven applications.
- We evaluate *Node.fz* using a diverse set of real-world Node.js applications, showing it increases the manifestation rate of concurrency errors with low overhead.

2. Background

In this section we define the EDA, discuss Node.js as the preeminent EDA framework, and explain the race conditions that can emerge in the EDA setting.

2.1 The Event-Driven Architecture

In its most common form, an EDA-based application has two main components, illustrated in Figure 1: a (typically single-threaded) event loop that processes incoming requests, and a worker pool to which it can offload expensive tasks. While there are other potential realizations, this Asymmetric Multi-Process Event-Driven (AMPED) architecture [41] is the one used by the mainstream general-purpose EDA frameworks: Node.js (JavaScript)[2] Twisted (Python)[3], EventMachine (Ruby)[4], libuv (C)[5], and Reactor (Java)[6].

Conceptually, EDA-based applications go through two phases: *registration* and *listening*. In the registration phase, the application defines callbacks to respond to different kinds of input (events). In the listening phase, the event loop

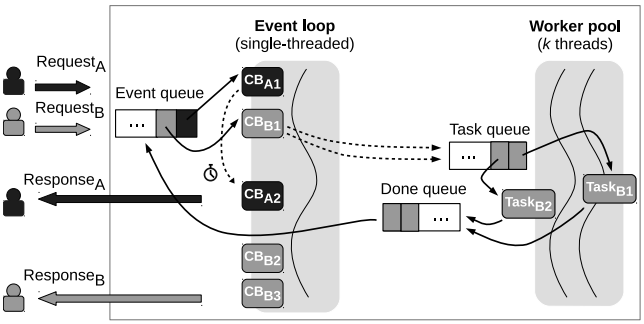

Figure 1. Event-Driven Architecture event loop and worker pool *à la* Node.js. The application is servicing requests from users A and B with callbacks (CBs). The callback chains for $Request_A$ and $Request_B$ are connected by lines. When these lines are dashed, the application has cooperatively partitioned the composition of its response. $Request_A$ is handled by two callbacks partitioned with a timer, while $Request_B$ offloads two tasks to the worker pool.

continuously awaits new events, executing the associated callback for each event.

The EDA has been shown to scale well compared to the One Thread Per Client (OTPC) architecture [42], though the jury is still out [53]. The essential trade-off is that of efficiency for reliability: in the OTPC architecture, each new client incurs more overhead (memory and context-switching), while in the EDA misbehaving clients have more opportunities to bring down vulnerable servers [16, 38].

2.2 Node.js, the Popular EDA Framework

Node.js [52] is an open-source EDA framework for developing server-side JavaScript applications. Its principal components are: (1) *libuv*, the core library providing an EDA with an event loop and a worker pool; (2) *Chrome V8* [2], a highly efficient JavaScript execution engine; and (3) C/C++ and JavaScript *libraries* abstracting functionality including the network, the file system, cryptography, and compression. Node.js has the largest package ecosystem of any language or framework [7], and as a result we consider it suitable for study as the most prominent example of an EDA system.

The Node.js worker pool is provided by libuv. The Node.js libraries use it to provide "asynchronous" file system I/O and DNS queries, and users can offload their own tasks to it too. Tasks for the worker pool are placed in its task queue and consumed concurrently by the workers. Workers signal the completion of tasks by placing the completed tasks on the worker pool's done queue and sending a "task done" event to the event loop. This process is illustrated in Figure 1: while handling user request $Request_B$, the callback CB_{B1} offloads two tasks to the worker pool. After the worker pool processes tasks $Task_{B1}$ and $Task_{B2}$, they are placed in the done queue, eventually triggering callbacks CB_{B2} and CB_{B3} to send the response to the user.

[2] See http://nodejs.org/.

[3] See http://twistedmatrix.com/.

[4] See http://rubyeventmachine.com/.

[5] See http://libuv.org/.

[6] See http://projectreactor.io/.

2.3 Programming and Race Conditions in the EDA

The primary programming style in the EDA is cooperative multitasking [48]. In the EDA, all incoming requests must pass through the event loop. If the callbacks associated with each request type compose their responses synchronously, pending requests will starve, especially when composing responses requires I/O or extensive computation. Consequently, the most basic rule of thumb of programming for the EDA is "never block the event loop" [12].

As hinted in Figure 1, to *avoid* blocking the event loop, the responses to requests should not be composed in a single heavy callback. Instead, response composition should be partitioned into multiple steps according to the principle of cooperative multitasking, resulting in the generation of intermediate events and callbacks to handle them. If developers follow this rule, they can create applications that offer both high responsiveness and high throughput.

Composing responses using this *callback chain* technique has important implications for software correctness: developers must provide guarantees for both *ordering* and *atomicity*. The EDA offers no guarantee of the order in which the event loop will process callbacks. For example, the callbacks associated with the expiration of a timer and the completion of a file system I/O may run in either order. Developers must therefore either write commutative callbacks or introduce their own ordering constraints. Furthermore, while each link in a callback chain is executed atomically by the (single-threaded) event loop, between any *pair* of links there is no guarantee of atomicity; the links from other callback chains can be interleaved.

Race conditions manifest when developers fail to acknowledge these issues. Most commonly, an ordering violation will frustrate the correct composition of even a single request (an intra-request race), while an atomicity violation impacts correctness when the system is processing multiple requests (an inter-request race). EDA-style race conditions are sensitive to the specific timing and order of events, making them difficult for developers to identify and reproduce. We visit these issues in detail in our concurrency bug study of real world open-source Node.js programs (§3).

Our bug study suggests an urgent need to provide programmers with solutions to explore broad swaths of the "event schedule space" in an effective manner. A remark from one of the developers inspired the approach we took to *Node.fz* (§4): *"Unfortunately, I'm not able to provide a simple test case because I dont know how to artificially expand the delay between the 'timeout' and 'close' events."*[7]. Using *Node.fz*, developers can do this and much more.

3. Concurrency Bug Study

This section provides the first, to the best of our knowledge, concurrency bug characteristic study of Node.js software. More generally, we believe it to be the first focused on server-side EDA concurrency errors. We have studied the patterns, manifestations, and fixes of concurrency bugs in real world open-source Node.js programs.

Name	Abbr.	Type	LoC	Dl/mo	Description
etherpad-lite	*EPL*	A	43K	N/A	Collaborative document editing
ghost	*GHO*	A	50K	4.5K	Blogging engine
fiware-pep-steelskin	*FPS*	M	8.2K	4	Policy enforcement point proxy
cinovo-logger-file	*CLF*	M	0.9K	111	Logging module
nes	*NES*	M	6.1K	6.8K	Native WebSockets for Hapi
agentkeepalive	*AKA*	M	1.9K	194K	keepalive http agent
webpack-tapable	*WPT*	M	0.4K	3.9M	Facilitates WebPack plugin use
socket.io-client	*SIO*	M	4.6K	4.9M	Real-time server framework
mkdirp	*MKD*	M	0.5K	23.3M	Recursive mkdir
kue	*KUE*	M	6.6K	69K	Priority job queue (w/ Redis)
restify	*RST*	M	5.5K	232K	Tool for RESTful APIs
mongoose	*MGS*	M	88K	969K	MongoDB-based object modeling

Table 1. Node.js software used in bug study. Abbr. (Abbreviations) are used throughout the paper. Type is A(pplication) or M(odule). Lines of Code (LoC) was computed using the cloc tool. LoC and Dl/mo (downloads/-month) are rounded. Statistics are as of February 2017. Sorted by application type and race type (see Table 2).

To identify bugs, we searched across all GitHub[8] bug reports for closed bugs in JavaScript-based projects that matched either "race" or "race condition"[9]. The search returned over 1000 results, from which we excluded race conditions in client-side JavaScript, as this type of race has been well studied in previous research [9, 35, 36, 43, 44]. From the remaining bugs, we manually selected 12 patched bugs for careful study, making our selection based on how well-documented the bugs were.

Table 1 shows a summary of the software whose bugs we studied, listing the program name, type (full-fledged Application or library Module), source code size in LoC, downloads in the past month, and a brief description. With a mix of applications and modules, a range of code base sizes, and a variety of purposes, we feel the selected software represents a broad range of Node.js practices. Hereafter, we will refer to software using its abbreviation.

The following sections describe a summary of our findings (§3.1), with in-depth descriptions of server-side EDA bug patterns (§3.2), manifestation (§3.3), and fixes (§3.4).

3.1 Summary of Findings

Our bug study reveals three key findings:

1. Like client-side JavaScript, server-side JavaScript software written for Node.js suffers from race conditions. We observed both atomicity violations and ordering violations in the races we examined, including a new subtype called commutative ordering violation.
2. Due to the "open system" nature of server-side software, we observed races on system resources like databases and

[7] See https://github.com/node-modules/agentkeepalive/issues/23.

[8] See https://github.com/.

[9] e.g. the search string "race language:JavaScript state:closed label:bug".

```
1 ...
2 this.sockets = [];
3 ...
4 Manager.prototype.socket = function (opts) {
5   var self = this;
6   ...
7   s = new Socket(self, opts);
8 -  socket.on('connect',function () {
9     if (notContains(self.sockets, s))
10      self.sockets.push(s);
11 -  });
12  ...
13 };
14
15 Manager.prototype.destroy = function (s) {
16   removeIfPresent(this.sockets, s);
17   if (this.sockets.length === 0)
18     this.close();
19   return;
20 };
```

Figure 2. Atomicity violation bug (*SIO*). The `destroy` method (line 16) can race with in-process connections (line 8)

the file system. This style of race has not been reported in client-side EDA (JavaScript) concurrency studies, and significantly complicates the task of anyone seeking to build a Node.js data race detector.

3. While the Node.js community has excellent techniques to fix the OV bugs in our study, they do not seem to have tools to help detect these or the AV bugs.

Table 2 summarizes our findings. The second column presents the GitHub bug (issue) number. Each bug has three major features: its general pattern, its specific manifestation, and the strategy employed to fix it. We identified two general patterns: atomicity violations (AV) and ordering violations (OV). With regard to bug manifestation, three columns indicate the events and object involved in the race, and the impact of the bug. Finally, the bug fix strategy is described in the last column. The final three rows describe the novel bugs we discovered. *FPS* (novel) is discussed in §3.2.2, while *SIO* (novel) and *KUE* (novel) are evaluated in §5.2.

3.2 Bug Patterns

In this section we introduce examples of AVs and OVs in server-side JavaScript applications, including a new subtype of OV called a commutative ordering violation (COV). We follow Lu et al. [29] and Hong et al. [24] in the definitions we use for AVs and OVs in the EDA context.

3.2.1 Atomicity Violations

The most frequent type of bug in our bug study was an AV. An AV occurs when two operations are intended to happen consecutively but another operation can be interleaved between them and affect the result. The other operation can

```
1 Job.prototype.markFailed () {
2     var self = this;
3   ...
4     if (self.canRetry) {
5 -   self.update().delayed();
6 +   self.update(function () {
7 +     self.delayed();
8 +   });
9   }
10    ...
11 }
```

Figure 3. Ordering violation (*KUE*). Both `update` and `delayed` are asynchronous. The `delayed` method must be called only after the `update` method completes.

occur before or after the relevant operations without issue, but not between them.

In event-driven programs, including Node.js, memory accesses in one event callback are executed without preemption, so AVs cannot occur *within* a callback. However, many concurrency bugs we found in this study are due to a false assumption of atomicity across callback chains. In the server-side EDA context, AVs tend to occur when the processing of one request can interfere with the processing of another request.

Many of the concurrency bugs in our study (9/12) were AVs. They had relatively little in common other than the shared bug type; the form of the AV and its effect varied widely from bug to bug. One example is illustrated in Figure 2, showing the patch to repair a (simplified) AV bug in the connection manager of *SIO*. Here we discuss the bug in the un-patched version. When a client requests a socket, the connection manager executes its `socket` method (line 4), creating a socket and adding it to its `sockets` array on the 'connect' event (lines 8–11). When the client disconnects, the connection manager executes its `destroy` method (line 15), deleting `s` from the `sockets` array (line 16) and closing the itself if there are no remaining connections (line 18).

Suppose a client attempts to connect to two different paths of the same server. If one connection completes quickly while the other takes a long time, the fast connection could be disconnected before the slow connection connects. In this case, the `destroy` method will find an empty `sockets` array, closing the manager and causing the slower connection to fail inappropriately.

3.2.2 Ordering Violations

Event-driven programs also suffer from OVs. An OV occurs when operation A should always be executed before operation B, but this order is not enforced.

OVs occur in the EDA when developers, overeager to partition the composition of responses, misunderstand the dependencies between their partitions and fail to enforce them. Therefore, OVs tend to occur during the processing of

Abbr.	Bug #	Race type	Racing events	Race on	Impact	Fix
EPL	2674	AV	NW-NW	Array	Crash (null dereference).	Check not null before use.
GHO	1834	AV	NW-NW	Database	Creates too many user accounts.	Deprecate functionality.
FPS	269	AV	NW-NW	Variable	Request hangs.	Fix incorrect control flow.
CLF	1	AV	FS-Call	Variable	Creates a duplicate file.	Rd/wr in the same callback.
NES	18	AV	NW-Timer	Variable	Crash (null dereference).	Check not null before use.
AKA	23	AV	NW-Timer	Variable	Throws error (possible crash).	Rd/wr in same callback.
WPT	243	AV	X-X	Variable	Throws error (possible crash).	Counter per request (callback chain).
SIO	1862	AV	NW-NW	Array	Request hangs.	Rd/wr in same callback.
MKD	2	AV	FS-FS	File system	Incorrect response (does not finish `mkdir`).	Check err code.
KUE	483	OV	NW-NW	Database	Job runs more than once.	Order async. calls using callbacks.
RST	847	(C)OV	FS-X	Array	Incorrect response (missing data).	Use an "async barrier".
MGS	2992	(C)OV	NW-NW	Database	Incorrect response.	Global counter.
SIO (novel)	PR 2721	AV	NW-Timer	Socket	Subsequent tests fail because the server's socket is occupied.	Disable automatic reconnection.
KUE (novel)	967	AV	*Unknown*	*Unknown*	Tests fail because lock is taken.	*Unknown*.
FPS (novel)	PR 339	(C)OV	NW-NW	Variable	Test case fails in wrong place.	Global counter.

Table 2. Characteristics of concurrency bugs in Node.js software, sorted by software type (Table 1) and race type. Race type is atomicity violation (AV) or ordering violation (OV); commutative ordering violations are marked with a (C). Races were either "solo" (intra-request) or due to competing concurrent requests. The racing events were network responses (NW) (typically from an external resource like a database), calls to the racy API (Call), timers (Timer), file system interactions (FS - uses worker pool), and "application-dependent asynchronous step" (X).

a single request, without the need of interference from other clients.

Several of the concurrency bugs in our study (3/12) were OVs. The patch to repair a (simplified) OV bug in *KUE* is shown in Figure 3. Here we discuss the bug in the un-patched version. This OV bug was caused by asynchronous status updates to a Redis database. On the failure of a job that can be retried later, the call to `update` sets the state of the job in the database to 'failed', while the call to `delayed` sets it to 'delayed'. Both `update` and `delayed` are asynchronous, launching concurrent updates to the status database.

The job's final state should be 'delayed', but because of the lack of ordering between the `update` and `delayed` methods, the job can end up with *two* states: both 'delayed' and 'failed'.

Commutative Ordering Violation The other two OVs in our study were of a sub-type of OV not previously reported in the literature. We call it a commutative ordering violation (COV). We suspect that it has gone unreported until now because it may occur more frequently in server-side EDA contexts than in client-side ones, due to the increased complexity of server-side applications.

Applications will sometimes launch multiple asynchronous requests, intending to run a callback only when all of them have completed. When the application prematurely runs this final callback, a COV bug occurs. While this is clearly a type of OV, it is distinctive because the ordering constraint is not between the asynchronous requests themselves, but rather in ensuring that they can execute in any order (commutatively) and that control will only shift to the final callback when appropriate.

Figure 4 shows the patch to repair a COV bug from *MGS*. The `firstStep` method (line 4) launches N `find` requests (line 7), each of which invokes the `nextStep` method (line 13) when complete. Whether each request is the final one is bound to the `nextStep` invocation. When the final request invokes `nextStep`, a promise is resolved (line 18) to indicate that `populate` (line 1) is complete.

The bug: there is no guarantee that asynchronous requests will complete in the same order in which they are submitted. The last launched `find` request may not be the last completed request, so the promise should be resolved using another mechanism.

This bug is similar to that of *RST*, in which an event callback makes a series of asynchronous `fs.read` calls, with callbacks updating a shared buffer. However, it returns prematurely, before all of the asynchronous reads have finished. The initial fix for *RST* used the same anti-pattern from Figure 4; the complete fix made use of an asynchronous barrier[10] instead. While studying the fix for the AV in *FPS*, we identified a novel COV in the associated test case[11] that uses the same anti-pattern, suggesting that this may be a common confusion even for professional Node.js developers.

3.3 Bug Manifestation Study

The findings of our bug manifestation study can be summarized as follows:

1. Events involved in race conditions stem from diverse sources such as network traffic, timers, user method calls, and the timing of worker pool work processing and "done" events (§3.3.1).
2. Race conditions are not only on shared memory (e.g. writes to variables and arrays), but also on system re-

[10] An asynchronous barrier is the EDA analogue of MPI's `MPI_Barrier` command.

[11] See our accepted pull request at https://github.com/telefonicaid/fiware-pep-steelskin/pull/339.

```
1  Model.prototype.populate = function (N) {
2    ...
3  + var remaining = N;
4    function firstStep (args, N) {
5      ...
6      for (var i = 0; i < N; i++) {
7        find(args,
8  -           nextStep.bind(this, i === N-1));[a]
9  +           nextStep.bind(this);
10     }
11   }
12   ...
13 - function nextStep (isLast, ...) {
14 + function nextStep (...) {
15     ...
16 -   if (isLast)
17 +   if (--remaining === 0)
18       promise.resolve(...);
19   }
20 }
```

Figure 4. Commutativity ordering violation (*MGS*). The `nextStep` function should only resolve the promise after *all* of the asynchronous `find` methods launched by `firstStep` have completed.

[a] `bind` creates a function that, when called, invokes the original function in the context and with the args provided.

sources (e.g. queries to database, I/O to file system) (§3.3.2).

3. Race conditions may result in severe consequences including server crashes and inconsistent database states (§3.3.3).

3.3.1 Racy Events

A brief evaluation of the events that triggered the races is informative. Races occurred in the callbacks for a diverse set of events, implying that detection or testing tools for server-side EDA applications in general, and Node.js applications in particular, must consider all these and more.

We were not surprised to find that many of the racy events (Table 2, column "Racing events") had to do with network traffic; this traffic was either between the client and the server (*EPL*, *FPS*, *NES*, *AKA*, *SIO*, *MGS*) or between the server and some back-end (e.g. to a Redis database server) (*GHO*, *KUE*). Of greater interest were the file system (*CLF*, *MKD*, *RST*) races, as these cannot occur in client-side JavaScript. Messiest of all was the *WPT* bug, because *WPT* is a plug-and-play framework and the racy events could have been any asynchronous task supported by Node.js.

3.3.2 What Were Races On?

By examining the types of objects on which the races occurred (Table 2, column "Race on"), we can see the kinds of racy accesses a data race detector for server-side JavaScript would need to detect.

Of course, just like client-side JavaScript, server-side Node.js programs have races on the property (variable, array, etc.) of some shared object (*EPL*, *FPS*, *CLF*, *NES*, *AKA*, *WPT*, *SIO*, *MKD*, *SIO*, *RST*). However, server-side software interacts with back-end systems like databases and the file system, and thus are vulnerable to race conditions on their state.

For example, *GHO* is vulnerable to a race on the state of its database. When a new username is registered, it asynchronously checks whether this username is already present in the database, and asynchronously adds it if it is not. Alas, if two `fetch` calls are interleaved and neither request finds a match, an extra username entry will be created.

The bug in *MKD* provides an example of a file system race. The `mkdirp` API works like the `mkdir -p` command: it creates a directory, creating any parents that don't already exist. In the *MKD* bug, two concurrent requests sharing the same prefix may race, causing one to return prematurely due to an incorrect handling of an EEXIST errno.

Unfortunately, these racy objects tell us that existing data race detectors developed for client-side JavaScript web applications [43, 44], which only consider object properties, cannot be directly applied to Node.js applications. They are defeated by these races on the state of external resources (the "open system" problem). Though attractive, modeling accesses to the shared resource file system or to a database as shared memory accesses does not strike us as a feasible extension: identifying a shared resource and determining conflicting requests to it (e.g., `fs.create` and `fs.unlink`) seems difficult in the Node.js context given the Node.js community's widespread reliance on external npm modules; there is no fixed set of system calls to instrument.

3.3.3 The Impact of Concurrency Errors

While concurrency errors in client-side JavaScript do not have particularly fearsome manifestations (e.g. unresponsive HTML buttons, an incorrectly initialized entry form, warnings written to a hidden console [35, 43, 44]), these 12 server-side EDA concurrency errors manifested in a variety of more serious ways (Table 2, column "Impact"). As in multi-threaded programs, impacts ranged from incorrect responses (3/12) all the way to potential server crashes (4/12). Coupled with the surging popularity of Node.js, the potential severity of errors emphasizes the need for tools to support server-side JavaScript developers.

3.4 Bug Fixes

In our bug fix study, we found that:

1. The AV bugs are solved in a variety of ways, most typically moving the intended-to-be consecutive accesses into the same callback (§3.4.1).
2. OVs can be solved using two semantically equivalent (but syntactically quite different) techniques: nested call-

backs, and the equivalent approaches of the async module and promises (§3.4.2).

3.4.1 Fixing Atomicity Violations

In multi-threaded programs, AVs are often fixed by lock-based mutual exclusion. Since the majority of these EDA-based AV bugs occurred in the event loop, and the event loop is single-threaded, each racy callback already is an atomic region: no locks required. As a result, the fixes frequently just moved the racy access from the later (asynchronous) callback into the initial callback, as shown in Figure 2 for the bug in *SIO*. The fixes for *AKA* and *CLF* follow the exact same fix strategy, and the fixes for *EPL* and *NES* are similar in spirit (testing for null).

An alternative approach we observed in the fix for *WPT* was to convert the shared (racy) variable into a variable local to each request (callback chain), eliminating potential interference between chains.

3.4.2 Fixing Ordering Violations

Though we analyzed only a small number of OV bugs, it seems that the fix strategy is well understood by the community. The fix for *KUE*'s OV bug illustrates one common pattern, and the fix for the OV bug in *RST* another.

Figure 3 shows the fix for the *KUE* OV. To ensure the order between events, `delayed` is invoked as a callback of `update`. This style matches that of the Node.js API, but taken to extremes can lead to deeply nested callbacks– "Callback Hell" [46].

Common ways to express more sophisticated ordering constraints are the async module[12] and the use of Promises (e.g. the Bluebird module[13]). In this vein, the COV bug in *RST* is fixed with an `async.barrier`, ensuring that all of the `fs.read` calls are completed before the next step. Bluebird's `Promise.all` API would also have served.

However, developers are also free to roll their own solutions, as shown in the patch for the COV bug in *MGS* (Figure 4). In the patch, the `remaining` counter is initialized with the number of requests `N`, and each asynchronous invocation of `nextStep` decrements it; the last completed callback is that for which `--remaining` is 0. We took the same approach for the *FPS* (novel) bug we repaired. The `async.barrier` and `Promise.all` APIs approaches are also suitable for addressing COV bugs.

3.4.3 Everybody Makes Mistakes

On a final note for the bug study, we want to emphasize that even developers familiar with effective EDA patterns still make mistakes. We do not believe that complex EDA-based software is significantly easier to get right than multi-threaded software, it just relies on a different paradigm.

The correct use of OV-preventing techniques does not protect against AVs. In *WPT*, the code affected by the bug made use of the `async` waterfall pattern, but when other callback chains were interleaved, it caused an AV. In *GHO* the same problem occurred, using promises instead of the async module.

More surprising, code that correctly used OV-preventing techniques still had OV bugs. In *MGS* (Figure 4), line 18 calls a promise, a typical ordering pattern, but *MGS* still had an OV! Clearly understanding ordering constraints is a non-trivial matter.

4. *Node.fz*: A Schedule Fuzzer for the EDA

The race conditions discussed in our bug study (§3) are difficult to find dynamically due to non-determinism in EDA-based systems like Node.js. This non-determinism, arising from the order in which inputs and intermediate events are handled by the event loop and the worker pool, masks the OVs and AVs that cause inter- and intra-callback chain races.

Inspired by the success of schedule fuzzing approaches to find race conditions in the multi-threaded context (e.g. [18]), we propose *Node.fz*, an EDA schedule fuzzing scheme designed for Node.js. *Node.fz* amplifies Node.js's internal non-determinism, allowing applications to explore a broader schedule space for the same input.

In this section we discuss the design and implementation of *Node.fz*. We first describe how Node.js works (§4.1), then evaluate the sources of non-determinism in the Node.js framework (§4.2), then discuss how we amplify this non-determinism using the techniques of de-multiplexing, event shuffling, and event delaying (§4.3), and conclude with a demonstration of the fidelity of *Node.fz* (§4.4).

4.1 How Node.js Works

During an application's listening phase, Node.js divides its time between checking for new events (using the libuv event loop) and executing and optimizing the associated JavaScript callbacks (using V8) .

When Node.js JavaScript code calls the asynchronous Node.js system call APIs, Node.js compiles the associated callbacks using V8 and registers the resulting function pointer with libuv. For example, when registering a listener on an `HTTPServer` object, Node.js asks libuv to monitor the associated socket and to invoke a function pointer when new data arrives. libuv tests this file descriptor on every iteration of its event loop (e.g. using `epoll` on Linux), executing the supplied callback with any data that arrives.

Each iteration of the libuv event loop examines in turn timers, pending callbacks, idle handles, prepare handles, I/O, timers again, check handles, and close callbacks. Timers are callbacks to be invoked after a certain amount of time has elapsed; pending callbacks finish work that was not quite completed on a previous iteration of the loop; idle, prepare, and check handles are callbacks to be invoked on every event

[12] See https://www.npmjs.com/package/async.

[13] See https://www.npmjs.com/package/bluebird.

loop iteration; I/O invokes callbacks registered in response to I/O events; close callbacks are invoked just before the associated objects are destroyed.

Node.js makes heavy use of the timer, closing, and I/O stages of the event loop. Node.js's use of the timer and closing stages is straightforward: Node.js translates JavaScript timers to libuv timers, and uses "closing" events to clean up the resources associated with JavaScript-level objects like HTTPServers. The I/O phase, on the other hand, is really a catch-all; network traffic, file system results, OS signal delivery, completed worker pool tasks, etc. are all implemented as I/O events, and these are the events that trigger most of the racy JavaScript callbacks from our bug study.

4.2 Non-Determinism in Node.js

Before we introduce *Node.fz*, this section first addresses the wide array of sources of non-determinism in Node.js, each of which will be fuzzed by *Node.fz*.

4.2.1 Non-determinism due to External Input

Input from external entities to an application is an obvious source of non-determinism. Node.js developers must be aware of the potential variations in input order from a broad range of sources.

Network traffic The order in which network traffic arrives is highly non-deterministic. While the traffic on a particular TCP socket is well-ordered, the traffic on UDP sockets and between multiple TCP sockets is not [45]. Applications cannot make assumptions about how many clients will make requests simultaneously, or about which client will make a request next.

Timers Using the `setTimeout` API, developers can queue a function to be invoked at least (and approximately) k milliseconds in the future. Timers are often used for ad-hoc synchronization, by deferring an action until a condition is met, and for timeouts, by aborting long-running operations.

There is significant non-determinism in the relative order of timer callbacks and other callbacks. This variation is due to changes in callback execution time, which varies based on the deployment conditions. For example, callback execution time will vary due to differing hardware or differing rate and type of incoming requests (e.g. leading to alternative V8 optimizations and file system caching).

Misc. As server-side applications, Node.js programs can make use of (and are therefore vulnerable to non-determinism in) a variety of features uncommon or unavailable in client-side JavaScript. For example, Linux Node.js applications can spawn child processes, send and receive UNIX signals, and do I/O to and monitor changes in the file system. In short, server-side JavaScript applications can be (and are) much more complex than client-side JavaScript.

4.2.2 Non-determinism due to Callback Chains

The EDA programming style discussed in §2.3 leads to a major source of non-determinism in Node.js applications. Developers typically structure the composition of responses into a callback chain, generally setting callback boundaries on I/O-bound activities (file system I/O, database queries, etc.). Though callback chains enable a responsive server with high throughput, they also expose applications to non-determinism: they can be interleaved in many different ways. The sources of non-determinism from external input (§4.2.1) are multiplied as callback chains are partitioned.

In Node.js, callback chain partitioning can be done on the event loop itself (e.g. using the `setImmediate` and `nextTick` APIs) or using the worker pool (calling libuv's `uv_queue_work` API from a C++ add-on). The `EventEmitter` pattern facilitates this style of code.

4.2.3 Non-determinism in the Worker Pool

The worker pool is the final source of non-determinism in Node.js applications, and a familiar one on the server side. Node.js applications can queue file system I/O requests, DNS-related queries, and user-defined tasks for asynchronous handling by the worker pool. The tasks in the worker pool queue are consumed concurrently by the workers. Once a worker completes a task, it places a corresponding "done" event on the event loop.

Both the size of each worker pool task and the scheduling of the worker pool workers affect the order in which the worker pool tasks are processed and their completion callbacks executed by the event loop. This variation in worker pool task processing and completion order leads to many possible schedules. Alternative orderings exist both within the worker pool (task processing and completion) and between the worker pool and the event loop (task processing and completion relative to incoming events in the event loop).

We provide one example of a possible worker pool race: concurrent I/O requests to the same file. The ext4 file system offers write atomicity only at the page granularity [15]. This means that if a Node.js application makes concurrent, overlapping, multi-page writes to a file, each affected page will consist of data from either write. File locks are not part of the native Node.js API, and this type of low-level race might be surprising to a developer from the client-side JavaScript perspective.

4.3 *Node.fz* Design

Having determined in our concurrency bug study (§3) that non-determinism in Node.js affects the manifestation of bugs, and having evaluated the sources of non-determinism in Node.js (§4.2), we now turn to the design of *Node.fz*.

At a high level, *Node.fz* takes control of the event loop's event queue and the worker pool's task and done queues. *Node.fz* then fuzzes these queues to explore alternative schedules. By shuffling the entries in the event queue before executing each callback, *Node.fz* yields schedules with alternative input and intermediate event arrival orders. By shuffling the entries in the worker pool's task and done

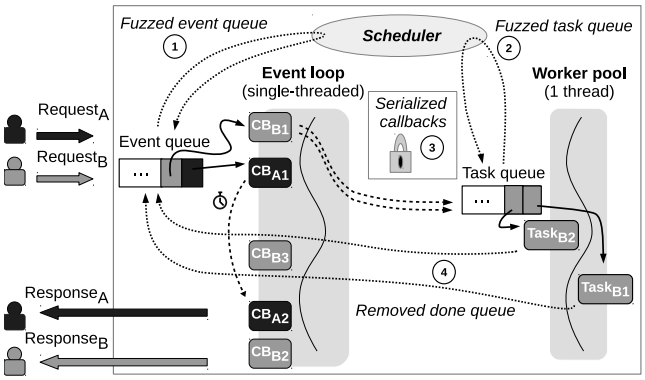

Figure 5. Highlights of *Node.fz*, our fuzzed EDA scheme, targeting AMPED architectures like Node.js. This figure illustrates the same case as Figure 1, with many callback orderings changed by the scheduler. Dotted lines indicate architectural changes compared to Figure 1.

queues, *Node.fz* produces schedules with alternative worker pool task processing and completion order.

Node.fz amplifies the non-determinism in Node.js using the techniques of de-multiplexing, event shuffling, and event delaying, achieving a greater exploration of the possible application schedule space without requiring any developer intervention. As a drop-in replacement for Node.js, developers can easily make use of *Node.fz* during development and test and then seamlessly switch to the optimized Node.js binary in production. Developers then have the assurance that their applications will be stable under a wider variety of deployment conditions.

4.3.1 Multiplexing

A recurring technique in the Node.js implementation is *multiplexing*, with the goal of minimizing the time it takes to complete an iteration of the event loop. Multiplexing application-level events into a single internal "wrapper" event reduces the total number of events handled by the event loop. This approach offers substantial performance gains, e.g. by reducing the number of system calls.

From a fuzzing perspective, however, multiplexing is undesirable. When we execute a "wrapper" event's callback, that (internal) callback consecutively executes a sequence of application-level callbacks. We want to be able to change the order of any pair of events, and multiplexing prevents us from interleaving other events into that consecutive list. Consequently, we eliminate multiplexing where possible.

In some cases, multiplexing is unavoidable. For example, when an `EventEmitter` emits an event, the callback registered for every listener is guaranteed by Node.js to be invoked successively, synchronously, and in registration order. Consequently, we cannot break this "wrapper" event into its constituent parts. We focus our attention, therefore, on cases where the use of multiplexing is not documented as part of the Node.js API: Node.js timers and the libuv worker pool done queue. In these cases, developers cannot assume any atomicity or ordering guarantees, even though the implementation currently provides them (§4.4 and §4.5).

4.3.2 Taking Control of the Event Loop

The racy events from the event loop identified during our bug study (see §3.3.1 and Table 2) were timers, I/O, and socket disconnects (which occur during the "closing" stage). Consequently, we insert hooks to the *Node.fz* scheduler (§4.3.4) when checking for expired timers, prior to handling ready file descriptors during the I/O phase, and prior to handling "closing" events. Hooks for the I/O phase are shown using dotted lines in Figure 5 (①).

4.3.3 Taking Control of the Worker Pool

Each worker in the libuv worker pool repeatedly takes a task from the queue, processes it, places it on the worker pool's "done queue", and signals the event loop. This signaling is implemented using a file descriptor included in the event loop's `epoll` set. When work is completed, a worker writes to this file descriptor, to be detected on the next pass through the I/O portion of the event loop. The event loop will then process every task in the done queue, so this internal file descriptor essentially multiplexes the done queue.

We take several steps to gain control of the worker pool. First, we serialize callback executions between the event loop and the worker pool, also effectively limiting the worker pool size to one. This allows the scheduler to be completely certain about the relative order of the execution of events and tasks, a fact on which we rely in §5.3. A drawback of doing so is that it eliminates the possibility of exposing several varieties of worker pool-related races (§4.2.3), though we did not identify any such races in our bug study (§3). Figure 5 illustrates this (③); unlike in Figure 1, no two callbacks ever execute at the same time (no horizontal overlap).

Second, we insert a hook to the *Node.fz* scheduler prior to taking an item from the work queue. The scheduler can then suggest which of the tasks the lone worker should handle next, simulating multiple workers. Note in Figure 5 (②) that the order of $Task_{B1}$ and $Task_{B2}$ are inverted compared to Figure 1, as may be suggested by the scheduler.

Third, we eliminate multiplexing of the done queue, for the reasons discussed in §4.3.1. To de-multiplex the done queue, we assign a private file descriptor to each task and add this file descriptor to the event loop's `epoll` set. When a task is completed, we write a byte to its file descriptor to signal the event loop that it is done. The individual task done callbacks can then be fuzzed by the scheduler just like any other I/O event, giving the scheduler complete control over the order in which done items are handled relative to each other and to other callbacks. In Figure 5 (④) you can see the effect this has: the order of the callbacks for $Task_{B1}$ and $Task_{B2}$, CB_{B2} and CB_{B3}, is inverted compared to

Figure 1, and CB_{A2} was able to run between them because they are no longer multiplexed.

4.3.4 *Node.fz* Scheduler

The *Node.fz* scheduler decides which pending events to handle and in what order. It exposes hooks for the event loop and the worker pool workers to call when they need to choose which events or tasks to handle. The scheduler has a number of parameters, outlined in Table 3.

Scheduling the event loop The event loop requests a scheduler decision when dealing with expired timers and with ready I/O descriptors. Included in the ready I/O descriptors are the done events in the de-multiplexed worker pool done queue (§4.3.3).

Expired timers are executed according to the timer deferral percentage, until one of them is deferred. After a timer is deferred, timer processing short-circuits until the next iteration of the event loop loop. Short-circuiting preserves the {timeout, registration time} timer callback ordering implemented in libuv. While this ordering is not documented by libuv or Node.js, it is assumed in several of the test suites we encountered in §5, and fuzzing it causes test failures. When deferring a timer, we also inject a delay of 5 milliseconds as a compromise between desiring forward progress and hoping for other events to arrive to interleave with the timer.

Once the event loop obtains the list of ready file descriptors from `epoll`, the scheduler shuffles them, moving each descriptor no further in the list than the shuffle distance ("`epoll` degrees of freedom") to allow a trade-off between extreme fuzzing and more realistic schedules. Each file descriptor is then handled or deferred according to the "`epoll` deferral percentage". This shuffling is illustrated in Figure 5: despite their arrival order, CB_{B1} is scheduled before CB_{A1}.

Scheduling the Worker Pool To maximize the fuzzing potential of the worker pool, the scheduler prompts the worker to wait until the task queue has at least "degrees of freedom" items in it, or until one of the "max delay" and "`epoll` threshold" limits is reached. The scheduler then selects one of the first "degrees of freedom" tasks in the queue at random for execution.

4.3.5 Implementation Details

We implemented *Node.fz* for Linux in roughly 10,000 lines of code, based on Node.js v0.12.7 (which used libuv v1.7.4). The changes we made to convert Node.js to *Node.fz* were entirely in the libuv event library. Though this introduced some limitations into the scope of our fuzzing (see §4.5), the reasons for this choice are twofold. First, the core event loop and worker pool reside in libuv, so placing our implementation here gives us full control over the event and worker pool schedule. Second, Node.js frequently releases new versions and; its source code is in a near-constant state of flux. So long as Node.js continues to rely on libuv as its event library, concentrating our efforts in the libuv insulates *Node.fz* from the rampant changes to the Node.js source. *Node.fz* can therefore easily be used in Node.js applications across a range of Node.js releases, as well as in other libuv-based software like Julia [4], MoarVM [6], and Luvit [5]).

We demonstrated the flexibility of our libuv-only approach by applying our libuv changes in three other branches of Node.js: two development branches, v3.x and v4.0.0-rc, and one release branch, v0.12.5-release. After substituting our version of libuv, we could compile and use *Node.fz* to say "hello world" in these different versions.

Though we only implemented *Node.fz* for Linux, extending our implementation to the other operating systems supported by libuv (Windows, OSX, etc.) would not be difficult.

4.4 *Node.fz* Fidelity

The *Node.fz* scheduler makes only legal fuzzing decisions according to the Node.js documentation:

1. **Fuzzing timers** Node.js does not provide an upper bound on how late a timer can be.
2. **Fuzzing `epoll` results** Fuzzing the ready file descriptors returned by `epoll` can be viewed from two perspectives. We are simulating either input arriving earlier or later than it actually did, or an `epoll` implementation that doesn't guarantee immediate notification of ready file descriptors. From either perspective such fuzzing is legal.
3. **Fuzzing the worker pool task queue** libuv offers no guarantee about the order of the handling of tasks.
4. **Fuzzing the worker pool done queue** libuv offers no guarantee about the order of the handling of done tasks relative to each other or to other events in the libuv event loop. It only assures the user that the completion callback of a task will be invoked only after its corresponding task has completed, a guarantee we also provide.

However, having a *legal* fuzzer is irrelevant if Node.js applications depend on undocumented implementation details of Node.js, or if Node.js is too tightly coupled to the libuv implementation. This is a legitimate concern, as an early version of *Node.fz* would also *shuffle* Node.js timers, which is legal but still caused some applications to fail . We next demonstrate that *Node.fz* is a *viable* alternative to Node.js by evaluating the Node.js test suite using *Node.fz*.

We evaluated *Node.fz* on the Node.js v0.12.7-release branch because it was the most recent branch that used the version of libuv on which we based our implementation. We compiled a non-fuzzy vanilla version ($\mathrm{node_V}$), then replaced the libuv component with our own and recompiled to obtain a fuzzy version ($\mathrm{node_{FZ}}$). We identified the test cases from the Node.js test suite that worked using $\mathrm{node_V}$, then evaluated them using $\mathrm{node_{FZ}}$.

Due to our implementation choices, $\mathrm{node_{FZ}}$ cannot accommodate concurrent access to libuv from Node.js. As a consequence, tests that make use of the debugger module (3 tests) and the VM module (2 tests) encounter a protective assert. Any application that relied on these modules would

Node.fz parameter name	Description	Standard parameterization
Event Loop: `epoll` degrees of freedom	Maximum shuffle distance of `epoll` ready items.	-1 (unlimited)
Event Loop: `epoll` deferral percentage	Probability of deferring a ready `epoll` item until the next iteration of the event loop.	10%
Event Loop: Timer deferral percentage	Probability of deferring an expired libuv timer until the next iteration of the event loop.	20%
Event Loop: "closing" deferral percentage	Probability of deferring a "close" event until the next iteration of the event loop.	5%
Worker Pool: Degrees of freedom	Work queue lookahead distance, i.e. number of simulated worker pool workers.	-1 (unlimited)
Worker Pool: Max delay	Total maximum time to wait to fill the worker pool work queue up to the degrees of freedom.	0.1 ms
Worker Pool: `epoll` threshold	Maximum time the event loop can be in `epoll` while we wait for the worker pool task queue to fill.	0.1 ms

Table 3. *Node.fz* scheduler parameters. The standard parameterization is described in §5.1.2.

also be immediately terminated. We did not encounter any such applications in our evaluation.

$\mathrm{node_{FZ}}$ passed all but one of the other tests without issue. It initially failed the test *test-fs-sir-writes-alot.js*, which atomically submits 10,240 file system requests and then waits for them to complete. As discussed in §4.3.3, to demultiplex the worker pool done queue we introduced one file descriptor per task. In the case of *test-fs-sir-writes-alot.js*, the event loop does not have the opportunity to close any of these file descriptors until every request has been submitted, concurrently consuming 10,240 file descriptors. $\mathrm{node_{FZ}}$ received `EMFILE` until we increased the limit on the test process's open file descriptor count using `ulimit`.

Based on the strength of the Node.js test suite, we conclude that $\mathrm{node_{FZ}}$ is a legal, viable alternative to Node.js.

4.5 *Node.fz* Limitations

Despite its success (§5), *Node.fz* has many limitations. Its primary constraints are:

1. *Node.fz* serializes callbacks (§4.3.3), degrading performance and limiting the possible races we can expose.
2. Our implementation was restricted to libuv for portability between versions of Node.js (§4.3.5). We could expose additional non-determinism were we to extend our implementation into the Node.js libraries (e.g. de-multiplexing Node.js timers).
3. As a dynamic tool, *Node.fz* can only identify races that can be exposed by the input to the software (e.g. data, test suite, etc.), so it may have false negatives. However, since *Node.fz* is a faithful alternative to Node.js (§4.4), it will not suffer from false positives.

5. Evaluation

Our evaluation seeks to answer the following research questions:

1. Does *Node.fz* improve the reproducibility of the bugs described in §3?
2. Does *Node.fz* uncover novel bugs?
3. How effectively does *Node.fz* explore the schedule space of an application?
4. What performance overhead does *Node.fz* introduce?

We ran all of our experiments on a machine with a 4-core Intel i7-4790 CPU (2 threads per core), 16GB RAM, running Linux 3.13.0-86. Due to the event-driven nature of the bugs in our study, however, we believe our experimental results are applicable to a wide variety of machine configurations.

5.1 Reproducing Bugs

Our primary research question was whether *Node.fz* increases the manifestation frequency of the race conditions from our bug study. We measured this by comparing the relative ability of Node.js ($\mathrm{node_V}$) and *Node.fz* ($\mathrm{node_{FZ}}$) to cause a bug to manifest. Due to the changes we made in libuv (§4.3.3), *Node.fz* will explore a slightly different area of the schedule space than Node.js even without fuzzing. As a result, we also measured the ability of non-fuzzed *Node.fz* ($\mathrm{node_{NFZ}}$) to cause a bug to manifest, choosing schedule parameters that induce no fuzzing (see Table 3).

5.1.1 Test Cases

Our bug study evaluated 12 concurrency bugs in Node.js software. In this and subsequent experiments we excluded those bugs whose reproduction we could not readily automate (*EPL*, triggered by web browser interaction) or that were not written in JavaScript (*WPT*)[14]. In the case of *GHO*, the bug report and the fix did not include enough clues to allow us to trigger the race externally, so we replicated the racy code in a small standalone application (*GHO′*) in Figure 6). The bugs in *KUE* and *RST* manifest frequently even using $\mathrm{node_V}$, so we only included *KUE* in our evaluation.

We drew test cases from the bug report where possible to increase the realism of our experiments. Where the bug report was too vague, we used the automated test case included with the commit where available. In cases where the commit did not include an automated test case, we developed a simple test of our own to imitate the actions described in the bug report. We observe that in 4/12 bug reports the patch did not include an automated test case; this was surprising, as in 3 of these 4 cases the associated GitHub project was well-established, with 2000-6500 commits.

The external test cases were all unit tests that could hit the bug with high or complete certainty on $\mathrm{node_V}$. Such cases often used timers to artificially encourage the manifestation

[14] The reproduce scenario for the *WPT* bug was written in CoffeeScript, and we could not successfully transpile it to JavaScript.

of the bug. These unit test-style test cases were retrospective, introduced after the discovery of the bug and deliberately targeting it by encouraging the racy path. In our view this approach is undesirable, as it over-tunes the test case to the implementation. While unit tests are better than nothing, also adding functional or system tests [32] that can uncover both the bug in question and other related bugs would be better practice. With this in mind, we therefore adapted the external test cases we used by introducing non-determinism (e.g. file system calls or timers) into the test to reduce the likelihood of hitting the bug, in effect converting these tests from unit tests to functional tests.

5.1.2 The Standard Parameterization

When we used $\mathrm{node_{FZ}}$ in this section, except where noted (§5.2.3) we used it with what we refer to as the "standard parameterization". This parameterization is a choice of fuzzing parameters that fuzzes each supported aspect of non-determinism in Node.js without perturbing the execution too dramatically. The values for the standard parameterization are listed in Table 3. We identified reasonable values using some synthetic races, and they proved effective across the spectrum of race conditions we set out to reproduce.

5.1.3 Experimental Results

We ran the test case used to reproduce each of the known bugs 100 times for each version of Node.js ($\mathrm{node_V}$, $\mathrm{node_{NFZ}}$, $\mathrm{node_{FZ}}$). We ran 100 tests because this is roughly the number of rounds of testing we ourselves use before declaring our own software "relatively bug free"; a tool that cannot cause a bug to manifest in 100 iterations is probably impractical. The results of this experiment are shown in Figure 6.

Overall, *Node.fz* was able to trigger the race conditions much more reliably than $\mathrm{node_V}$. The variation in bug reproduction rates for different modules is due to factors like how difficult the bug is to hit in general, how effective the (adapted or hand-crafted) test case in question is at triggering the race, and how relevant the standard parameterization (§5.1.2) is in each case. We note that only the *KUE* and *FPS* bugs manifested using $\mathrm{node_V}$; the rest could only be detected using $\mathrm{node_{FZ}}$. In some cases, $\mathrm{node_{NFZ}}$ was sufficient to trigger the races as well, but was generally inferior to $\mathrm{node_{FZ}}$. Overall, the use of even the generic standard parameterization clearly offers a marked improvement in bug reproduction, indicating that it will also increase the rate of novel bug manifestation.

5.2 Finding Novel Bugs

We searched for novel bugs by running the full test suites of the software whose bugs we studied. We found a total of three bugs (two novel) across two of the modules, *SIO* and *KUE*. The manifestation rate of these bugs is also shown in Figure 6.

These manifestation rates are based on 50 iterations rather than 100 because running a full test suite can be far more time consuming than running individual tests, even when using $\mathrm{node_V}$. Because we used the most recent version of each software's test suite in this experiment, we had to omit *NES* and *GHO*, whose most recent versions are no longer compatible with the version of Node.js (libuv) on which *Node.fz* is based.

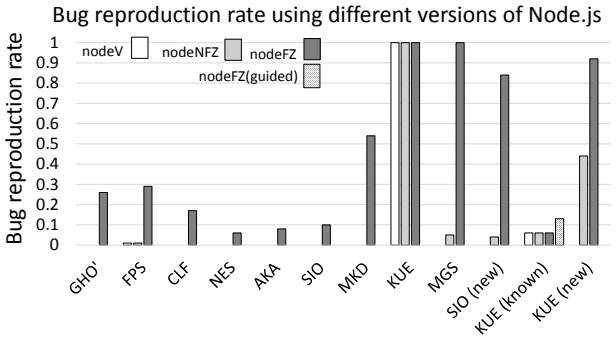

Figure 6. Bug reproduction rates. In the majority of these cases, only $\mathrm{node_{FZ}}$ was able to cause the bug to manifest.

5.2.1 Novel bug in socket.io (c94058f9)

We identified a novel atomicity violation in the *SIO* test suite. $\mathrm{node_{FZ}}$ uncovered a test case that failed to clean up one of its client requests, which was on a repeating timer. When the timer expired, it would attempt to connect to a server shared by all of the test cases. If it happened to wake up during the small subset of sensitive test cases, it would steal a connection and cause those cases to time out. This bug manifested far more frequently using $\mathrm{node_{FZ}}$ than using $\mathrm{node_V}$. Our patch for this issue was accepted[15].

5.2.2 Novel bug in kue (4c5711ba)

We identified a novel bug in the *KUE* test suite. One of the test cases failed regularly using both $\mathrm{node_{NFZ}}$ and $\mathrm{node_{FZ}}$. We traced the cause of the failure to a timeout due to an inability to promptly acquire a lock from Redis, suggesting a deadlock. Though we couldn't identify the root cause of the issue, we have contacted the maintainers with a description[16].

5.2.3 Guided Fuzzing Increases Reproduction Rate

We independently identified a bug in the 2014 version of the *KUE* test suite (03736bd7) that had since been fixed. The test suite assumed that a timer would not be executed with high precision, crashing if a timer went off too soon after its scheduled deadline. It manifested in 3/50 trials when running the test suite on $\mathrm{node_V}$, $\mathrm{node_{NFZ}}$, and $\mathrm{node_{FZ}}$.

The failed assertion said a timer had gone off early. Accordingly, we tweaked the fuzzing parameters to favor accurate timers; deferring worker pool tasks and event loop

[15] See https://github.com/socketio/socket.io/pull/2721.

[16] See https://github.com/Automattic/kue/issues/967.

events with high probability caused the event loop to spend most of its time spinning instead of executing callbacks. This in turn meant that it could identify and execute ready timers relatively quickly. Our first tweak to the parameterization quadrupled the manifestation rate to 13/50; a higher reproduction rate simplified our subsequent root cause analysis.

We observe that this bug is neither an AV nor an OV as described in §3. Rather, this bug is a "race against time"; the assert is simply that the time of callback execution is at least k milliseconds after the time of registration.

5.2.4 Pros and Cons of this Approach

Our approach to identifying new bugs ably demonstrates both the strengths and the weaknesses of *Node.fz*. On one hand, *Node.fz* is easily used with existing Node.js software and test suites, and *Node.fz* was able to expose races in the test suite or the software more able than $\mathrm{node_V}$. As a runtime approach, *Node.fz* requires no expertise in the software under test. On the other, however, as a dynamic tool, *Node.fz* can only increase the manifestation rate of race conditions exposed by the test suite. In essence, *Node.fz* increases the power of the existing test suite to expose bugs, but it cannot infer bugs that the test suite could never expose.

We believe that we discovered relatively few novel bugs for three reasons. First, the software we studied is relatively mature, so many race conditions have already been addressed. Second, without expertise on each piece of software, we could only report bugs that caused a crash or a test failure; others may have gone unnoticed. Third, manual inspection of the suites suggested that tests are typically unit tests rather than functional or system tests, and we feel that the latter types of tests are more likely to expose race conditions in software.

5.3 Schedule Space Exploration

In §5.1 and §5.2, we demonstrated the practicality of *Node.fz*. To determine its generality, we measured the variation in the schedules *Node.fz* explores when executing the test suites of some of the modules identified in our bug study.

We define a *Node.js schedule* as the order in which JavaScript callbacks are executed and the worker pool operations are interleaved. We define a *libuv schedule* as the order in which libuv callbacks are executed and the worker pool operations are interleaved. Note that at the libuv level, we cannot accurately identify the Node.js schedule because the callbacks supplied to libuv are black boxes; we do not bridge the semantic gap [14].

The greater the schedule variability, the more likely race conditions are to manifest. Since *Node.fz* is implemented at the libuv level, we propose a simple measure to approximate the libuv schedule variability; this is in turn an approximation of the Node.js schedule variability. We record the *type* (e.g. "timer", "network read", "worker pool task") of each libuv callback as we execute it; the resulting *type schedule* approximates the libuv schedule[17]. The variation between two libuv type schedules can be measured using the Levenshtein Distance (LD) [28] (string edit distance)[18].

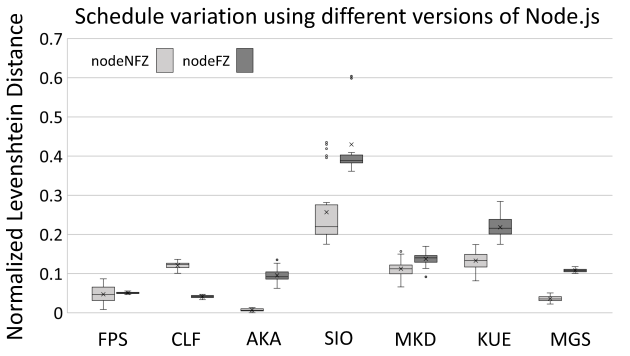

Figure 7. Normalized Levenshtein Distance between the type schedules generated by running the test suites of the indicated modules 10 times using $\mathrm{node_{NFZ}}$ and $\mathrm{node_{FZ}}$. Note that an LD of 1.0 would occur only when the two type schedules have nothing in common, not something we expect to see here.

Figure 7 shows the result of the pairwise LD between the type schedules produced by 10 executions of the test suites for some of the modules from our bug study using $\mathrm{node_{NFZ}}$[19] and $\mathrm{node_{FZ}}$. We normalize the LD for each module against the maximum possible value so that the variation between schedules can be compared across modules. Due to the computational complexity of the Levenshtein Distance algorithm, we considered only the first 20K callbacks from each schedule. This truncated the schedules from *FPS*, *CLF*, *SIO*, and *MGS*, which had 66K, 210K, 37K, and 56K callbacks per execution, respectively.

In every case but CLF, $\mathrm{node_{FZ}}$ increased the schedule variation, in most cases appreciably or significantly. We believe the significant truncation of the *CLF* schedule led to the surprising decrease in schedule variation for that test. Given the approximate nature of the type schedules we used, this experiment indicates, albeit imprecisely, that *Node.fz* expands the schedule space explored by a test suite.

5.4 Performance Evaluation

To determine the amount of overhead induced by *Node.fz*, we evaluated the running time of the test suites for recent versions of some of the buggy modules, while being run using $\mathrm{node_V}$, $\mathrm{node_{NFZ}}$, and $\mathrm{node_{FZ}}$. Figure 8 shows the normalized time to run the test suite under the various versions.

[17] The type schedule is not an exact schedule because it cannot differentiate between alternative orderings of two callbacks of the same type. For example, if the order of two timers were inverted, the corresponding type schedules would be identical.

[18] The LD answers the question, "How many steps are required to turn one string into the other?"

[19] $\mathrm{node_{NFZ}}$ is as close an emulation of $\mathrm{node_V}$ as possible while still serializing callbacks to produce a comparable type schedule.

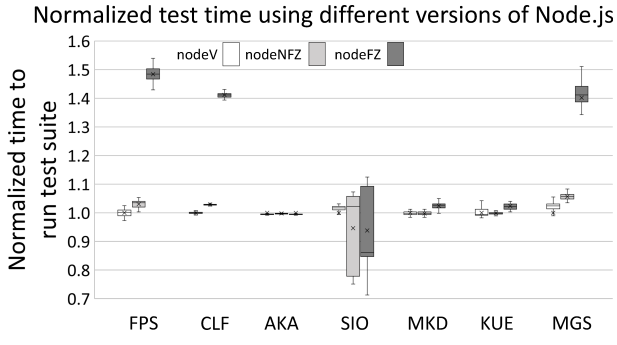

Figure 8. Normalized performance overhead to run the test suite of the indicated modules using $\mathrm{node_V}$, $\mathrm{node_{NFZ}}$, and $\mathrm{node_{FZ}}$. Each suite was run 50 times on an otherwise idle system.

Overall the results are encouraging. Though even a vanilla parameterization of *Node.fz* introduces overhead due to the callback serialization, from the comparable performance of $\mathrm{node_V}$ and $\mathrm{node_{NFZ}}$ it is clear that our changes to libuv did not introduce appreciable overhead in these cases. The increased overhead using $\mathrm{node_{FZ}}$ (up to ~1.5x) is presumably due to the delays we inject. The amount of overhead will vary with different choices of scheduler parameters.

6. Discussion and Related Work

In this section we discuss the relationships between this paper and previous work in bug studies and test aids for multi-threaded programming and for client-side JavaScript.

Bug Studies The largest concurrency bug study to date was on multi-threaded programs [29], and we are indebted to Lu et al. for their careful definitions of AVs and OVs. However, as we discussed in §3, the forms that AVs and OVs take in the EDA context are unique and also worthy of study. While there have been studies of JavaScript bugs [35, 36], these studies have not examined in detail the root causes and fix patterns in the way that we have done.

Schedule exploration Schedule exploration has been applied in the multi-threaded context by injecting random or guided variation into thread schedules (e.g. [18, 20, 39, 47, 49]). To the best of our knowledge, *Node.fz* is the first to extend this notion into the realm of the server-side EDA. *Node.fz* focuses on the schedule of events, not threads, and takes a randomized approach suited to long-lived server processes.

Though systematic testing of multi-threaded [19, 34] and "asynchronous reactive" [17] programs has been proposed, randomized scheduling has been shown to be just as effective [51], and we also found randomized schedule fuzzing to be effective in the EDA. Because it controls all points of non-determinism in Node.js, *Node.fz* can also enable more systematic exploration of Node.js application schedules.

Client-side JavaScript Other researchers have discussed aids to detect bugs in client-side JavaScript [27, 43, 44]. Though the prevalent client-side and server-side JavaScript environments are all event-driven, these client-side analyses are tuned to the relationship between JavaScript and the browser's DOM rather than to the relationship between JavaScript and the "open system" (e.g. the file system, a database, etc.). Our bug study shows that these solutions cannot be easily applied to Node.js applications, primarily due to the open system nature of Node.js and the concomitant race conditions, and in part due to scalability issues, as server-side applications are much longer lived.

Node.js tools We are aware of two related tools in the realm of Node.js. Madsen et al. presented a static analysis using the event-based call graph [30], though they apply it to bugs more common in a novice's program than in an expert's. In contrast, *Node.fz* can expose bugs even in large, well-maintained Node.js projects. In the broader Node.js community, the node-mocks project[20] enables a narrow form of JavaScript-level schedule fuzzing, and is subsumed by *Node.fz*.

Android The Android environment is another hotbed of event-driven programming, and researchers there have proposed several dynamic data race detectors [11, 25, 31] and record-and-replay systems [26]. These tools are tailored to the Android system architecture, and cannot easily be ported to the Node.js architecture.

Misc. Lastly, like *Node.fz*, Chadha et al. [13] peek ahead into the EDA event queue, though they do so to prime caches rather than to shuffle the order of events.

7. Conclusion

This paper presents *Node.fz*, a novel schedule fuzzing test aid for server-side EDA programs, targeting the Node.js environment. The design of *Node.fz* was based on the first concurrency bug study of real-world Node.js (and EDA) software, in which we discussed the forms atomicity and ordering violations take in the EDA, and draw attention to a common sub-type of ordering violation which we term a commutative ordering violation. Based on the root causes of the bugs in our study, we designed *Node.fz* to shuffle the order of input events and callback chains as they appear in the Node.js runtime. Our results show that *Node.fz* can trigger known bugs more frequently, expose new bugs, and expand the schedule space explored by a test suite, all with an acceptable overhead.

Acknowledgments

We appreciate the efforts of Talha Ghaffar and M. Usman Nadeem in our search for novel bugs. Ayaan Kazerouni and Gregor Kildow offered helpful criticism on drafts of the paper. We are grateful to the anonymous reviewers and to our shepherd, Zheng Zhang, for their thoughts and guidance.

[20] See https://github.com/vojtajina/node-mocks.

References

[1] Cylon.js. https://cylonjs.com.

[2] Chrome V8. https://developers.google.com/v8/.

[3] Node-RED. https://nodered.org/.

[4] Julia. http://julialang.org/.

[5] Luvit. https://luvit.io/.

[6] MoarVM : A 6model-based VM for NQP and Rakudo Perl 6. https://github.com/MoarVM/MoarVM.

[7] Module Counts. http://www.modulecounts.com.

[8] Node.js. https://nodejs.org/en/.

[9] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 367–377, 2014.

[10] A. Allan. *Learning iPhone Programming*. O'Reilly Media, 2010.

[11] P. Bielik, V. Raychev, and M. Vechev. Scalable Race Detection for Android Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 332–348, 2015.

[12] M. Casciaro. *Node.js Design Patterns*. 1 edition, 2014. ISBN 9781783287314. doi: 10.1002/ejoc.201200111.

[13] G. Chadha, S. Mahlke, and S. Narayanasamy. Accelerating Asynchronous Programs Through Event Sneak Peek. In *Proceedings of the Forty-Second International Symposium on Computer Architecture (ISCA)*, pages 642–654, 2015.

[14] P. M. Chen and B. D. Noble. When Virtual is Better Than Real. *Hot Topics in Operating Systems (HotOS)*, 3:116–121, 2001.

[15] L. Czerner. ext4: Make Reads/Writes Atomic With i_rwlock semaphore - Patchwork. https://patchwork.ozlabs.org/patch/91834/.

[16] J. Davis, G. Kildow, and D. Lee. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *Proceedings of the Tenth European Workshop on System Security (EuroSec)*, page 6, 2017.

[17] A. Desai, S. Qadeer, and S. Seshia. Systematic Testing of Asynchronous Reactive Systems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2015.

[18] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[19] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-Bounded Scheduling. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (PoPL)*, 2011.

[20] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the Eleventh USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] J. Governor, D. Hinchcliffe, and D. Nickull. *Web 2.0 Architectures*. O'Reilly Media / Adobe Developer Library, 2009.

[22] J. Harrell. Node.js at PayPal, 2013. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.

[23] A. T. Holdener. *Ajax: The Definitive Guide*. O'Reilly Media, Inc., 2008.

[24] S. Hong, Y. Park, and M. Kim. Detecting Concurrency Errors in Client-side JavaScript Web Applications. In *Proceedings of the Seventh International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[25] C.-H. Hsiao, Y. Jie, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race Detection for Event-Driven Mobile Applications. In *Proceedings of The Thirty-Fifth Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[26] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet Lightweight Record-and-Replay for Android. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 349–366, 2015.

[27] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless Model Checking of Event-Driven Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[28] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[29] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning From Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[30] M. Madsen, F. Tip, and O. Lhoták. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 505–519, 2015.

[31] P. Maiya, A. Kanada, and R. Majumdar. Race Detection for Android Applications. In *Proceedings of The Thirty-Fifth Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[32] S. McConnell. *Code Complete*. Pearson Education, 2004.

[33] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura. *Programming Android*. O'Reilly Media, 2012.

[34] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical report, Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[35] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *Proceedings of the Seventh International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64, 2013.

[36] F. S. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript Errors in the Wild: An Empirical Study. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering (ESEM)*, pages 100–109, 2011.

[37] J. O'Dell. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app, 2011. http://venturebeat.com/2011/08/16/linkedin-node/.

[38] A. Ojamaa and K. Duuna. Assessing the Security of Node.js platform. In *Proceedings of the Seventh International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 348–355, 2012.

[39] B. K. Ozkan, M. Emmi, and S. Tasiran. Systematic Asynchrony Bug Exploration for Android Apps. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 455–461, 2015.

[40] S. Padmanabhan. How We Built eBay's First Node.js Application, 2013. http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/.

[41] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999.

[42] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the Performance of Web Server Architectures. In *Proceedings of the Second European Conference on Computer Systems*, volume 41, pages 231–243. ACM, 2007.

[43] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race Detection for Web Applications. In *Proceedings of The Thirty-Third Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[44] V. Raychev, M. Vechev, and M. Sridharan. Effective Race Detection for Event-Driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[45] B. Rieken and L. Weiman. *Adventures in UNIX Network Applications Programming*. John Wiley & Sons, Inc., 1992.

[46] S. Robinson. Avoiding Callback Hell in Node.js. http://stackabuse.com/avoiding-callback-hell-in-node-js/.

[47] K. Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of The Twenty-Eighth Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[48] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[49] S. D. Stoller. Testing Concurrent Java Programs Using Randomized Scheduling. In *Electronic Notes in Theoretical Computer Science*, 2002.

[50] R. E. Sweet. The Mesa Programming Environment. *ACM SIGPLAN Notices*, 20(7):216–229, 1985.

[51] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency Testing Using Schedule Bounding: An Empirical Study. In *Proceedings of the Nineteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[52] S. Tilkov and S. V. Verivue. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[53] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.