A Comprehensive Study on Real World Concurrency Bugs in Node.js

Jie Wang[1,2], Wensheng Dou[1,*], Yu Gao[1,2], Chushu Gao[1], Feng Qin[3], Kang Yin[1,2], Jun Wei[1,2]
[1]State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[2]University of Chinese Academy of Sciences, China
[3]Dept. of Computer Science and Engineering, The Ohio State University, United States
[1]{wangjie12, wsdou, gaoyu15, gaochushu, yinkang15, wj}@otcaix.iscas.ac.cn, [3]qin.34@osu.edu

***Abstract*—Node.js becomes increasingly popular in building server-side JavaScript applications. It adopts an event-driven model, which supports asynchronous I/O and non-deterministic event processing. This asynchrony and non-determinism can introduce intricate concurrency bugs, and leads to unpredictable behaviors. An in-depth understanding of real world concurrency bugs in Node.js applications will significantly promote effective techniques in bug detection, testing and fixing for Node.js.**

In this paper, we present *NodeCB*, a comprehensive study on real world concurrency bugs in Node.js applications. Specifically, we have carefully studied 57 real bug cases from open-source Node.js applications, and have analyzed their bug characteristics, e.g., bug patterns and root causes, bug impacts, bug manifestation, and fix strategies. Through this study, we obtain several interesting findings, which may open up many new research directions in combating concurrency bugs in Node.js. For example, one finding is that two thirds of the bugs are caused by atomicity violation. However, due to lack of locks and transaction mechanism, Node.js cannot easily express and guarantee the atomic intention.

***Index Terms*—JavaScript, Node.js, event-driven, concurrency bug, empirical study.**

I. Introduction

JavaScript has become one of the most popular programming languages. According to the recent surveys from Stack Overflow [1] and Node.js foundation [2], JavaScript is quickly surpassing the popularity of other back-end programming languages (e.g., PHP and Java). As a server-side framework, Node.js, which is built on Google's V8 JavaScript engine [3], is becoming a popular platform for server-side applications. One prominent evidence is that, the Node.js' default package repository, *npm* [4], has become the largest package registry in the world, and contains over 400,000 available packages, doubling the next most popular package registry (the Apache Maven repository) [5].

In order to optimize throughput and scalability of server-side applications, Node.js adopts an event-driven architecture, which is capable of asynchronous I/O. Thus, developers can create highly scalable server-side applications without using threading. However, asynchronous I/O and non-deterministic event processing in Node.js can introduce intricate concurrency bugs. These concurrency bugs can result in unreliable and unpredictable application behaviors, such as crashes, and inconsistent states in databases and files.

Existing studies have focused on concurrency bugs in multi-threaded systems [6], distributed systems [7], Android and client-side JavaScript applications [8], and also proposed many interesting testing and analysis techniques [9][10][11][12]. The concurrency bugs in Node.js differ from those in traditional systems as they originate from different programming paradigms and execution environments. First, multi-threaded systems and distributed systems focus on concurrency bugs related to multi-thread [6] or untimely external events [7], while Node.js atomically processes each event one by one in a single thread, without interruption. Second, the concurrency in Android mostly concerns the Android GUI model and asynchronous tasks executed in other threads [10][11], while Node.js does not have these features. Third, existing studies on concurrency bugs in client-side JavaScript applications mostly focus on the features like DOM and AJAX [8][12], while Node.js does not have DOM and AJAX. In contrast, Node.js has the ability to access system resources, e.g., databases and files.

Node.js is relatively new, and little is known about concurrency bugs in Node.js applications. Whether existing studies and techniques [6][7][8][10][11][12] are applicable to Node.js applications remains an open question. Therefore, we aim to bridge this gap by conducting a comprehensive study on such concurrency bugs in Node.js applications. We believe a deep understanding of real world concurrency bugs will significantly promote effective techniques in concurrency bug detection, testing, and fixing in Node.js.

In this paper, we present *NodeCB*, the first (to the best of our knowledge) comprehensive real world concurrency bug study in Node.js applications. By using the keyword-based searching, we obtain 1,583 concurrency- and JavaScript- related bug reports in GitHub. Then we check these bug reports and finally collect 57 real world concurrency bugs in 53 open-source Node.js projects. We thoroughly study these 57 concurrency bugs, and try to answer the following research questions:

- **RQ1 (Bug patterns and root causes):** What are common bug patterns of concurrency bugs in Node.js? What are their root causes?
- **RQ2 (Bug impacts):** Do concurrency bugs have severe failure symptoms? What impacts do they have in Node.js applications?

[*] Corresponding author

- **RQ3 (Bug manifestation):** How do concurrency bugs in Node.js manifest themselves? How are they triggered, e.g., the timing condition, and input conditions?
- **RQ4 (Bug fix strategies):** How do developers fix concurrency bugs in Node.js in practice? Are there any common fix strategies?

Through investigating the above four research questions, we made many interesting findings. The main ones are:

- Concurrency bugs in Node.js can be caused by atomicity violation (65%), order violation (30%), and starvation (5%). Although each event is guaranteed to be processed atomically by Node.js, atomicity across multiple events cannot be properly enforced without lock or transaction mechanisms. Furthermore, existing work mostly focuses on order violation in event-driven applications (e.g., client-side JavaScript [8][12] and Android[10][11][13]). This suggests that more research should be conducted on atomicity violation in Node.js.
- Most concurrency bugs contend against shared variables (54%), databases (26%) and files (14%). This suggests that, besides shared variables [10][11], concurrency bug detection approaches should pay more attention to shared resources like databases and files.
- APIs in Node.js are written in an asynchronous and event-driven way. They usually have unclear API protocols (e.g., event order and atomicity specifications). 28 (49%) of concurrency bugs are caused by API misuse, indicating that developers may misunderstand the specifications of asynchronous APIs.
- Almost all (93%) concurrency bugs in Node.js are triggered by enforcing orders among no more than 4 events. This indicates that exploring possible orders among every small group of events can test Node.js concurrency efficiently.
- Almost all the studied concurrency bugs cause severe failures, including crashes/exceptions (33%), hangs (7%), incorrect database/file states (32%), wrong outputs (11%) and other operation failures (17%).
- Most concurrency bugs were fixed by a small set of fix strategies. But, only a small portion (23%) of bugs were fixed by simply adding synchronization (e.g., nested callbacks), which is the main approach used by existing automated concurrency bug fixing approaches [14]. This indicates that further automated bug fixing approaches are needed.

In summary, we make the following contributions:

- We conduct the first comprehensive study of real world concurrency bugs in Node.js applications. Our findings can help better understand concurrency bugs in Node.js applications, and provide guidelines to this topic.
- Our 57 documented concurrency bugs can serve as a basis for future work on finding and fixing them. We have made the collected bugs available online for future studies (http://www.tcse.cn/~wsdou/project/NodeCB).

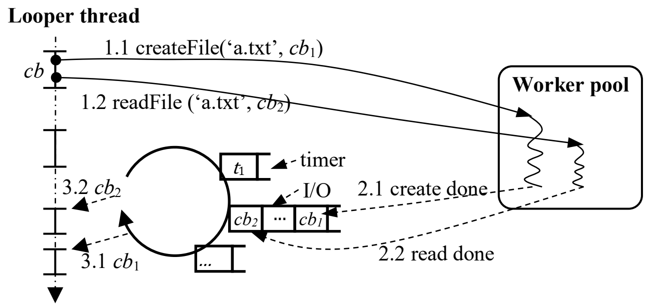


Fig. 1. The event-driven model in Node.js. *The solid arrows denote the invocation of an asynchronous operation (1.1 and 1.2), the dotted arrows denote that the operations are completed (2.1 and 2.2) and the corresponding callbacks are invoked (3.1 and 3.2). The looper thread executes each callback one by one, without interruption.*

II. Background

In this section, we first introduce the event-driven model in Node.js, and then explain the sources of non-determinism in this model.

A. Event-Driven Model in Node.js

The event-driven model in Node.js is shown in Fig. 1. It mainly consists of two parts: a single *looper thread* and a *worker pool*. Both parts are supported by *libuv* [15] in Node.js. The looper thread is responsible for executing user code, including callbacks that are defined to respond to events. For some expensive tasks, e.g., file operations, Node.js offloads them to the worker pool, and executes them asynchronously. By offloading expensive tasks to the worker pool, the looper thread would not be blocked by these expensive tasks.

In Node.js, the basic event processing flow is described as follows: (1) The looper thread fetches an event in the event loop, and executes its callback *cb*. (2) If the callback *cb* invokes an asynchronous I/O operation, the looper thread will offload it to the worker pool, and register a new callback associated with the asynchronous I/O operation. For example, in Fig. 1, callback *cb* offloads two I/O operations *createFile ('a.txt', cb_1)* and *readFile ('a.txt', cb_2)* to the worker pool, and uses cb_1 and cb_2 as their callbacks, respectively (steps 1.1 and 1.2). Thus, the looper thread can proceed with other events in the event loop, instead of waiting for the completion of these two operations. (3) The worker pool can use a worker thread to perform each asynchronous I/O operation, and put an event into the event loop when the operation is done. For example, the worker pool puts two events *'read done'* and '*create done*' into the event loop (steps 2.1 and 2.2). (4) The looper thread fetches these two events one by one sometime later, and executes their callbacks cb_2 and cb_1 (steps 3.2 and 3.1). The Node.js application continues the above steps until the application exits. Note that, each callback is guaranteed to be executed without interruption.

In Node.js, the event loop consists of seven FIFO event queues that hold different types of events: *timer*, *I/O*, *pending*, *idle*, *prepare*, *check*, and *close* [16]. For example, all events scheduled by *setTimeout* will be put into the *timer* event queue, while I/O events will be put into the *I/O* event queue. The looper thread in turn processes these seven event queues in a round-robin manner: when a queue has been exhausted or the amount of the invoked callbacks for a queue reaches a given threshold, the looper thread will move on to the next queue. Although the events in each queue are scheduled in order, the events across queues can be scheduled non-deterministically. For example, we cannot know the processing order of a *timer* event and an *I/O* event. Worse, the looper thread may register high-priority callbacks first. For example, a callback scheduled by *process.nextTick* will be executed immediately after the current callback completes, and the I/O events will be processed before moving on to process the events set by *setImmediate*.

B. *Non-determinism in Node.js*

Node.js applications make heavy use of asynchronous operations, and further handle these operations' results by their callbacks. Although the looper thread atomically processes each event one at a time without interruption, Node.js applications still suffer from various non-determinism. We summarize the sources of non-determinism in a single Node.js process into the following three phases according to the time when it occurs.

Non-deterministic execution of asynchronous operations. When multiple asynchronous tasks are delegated to the worker pool simultaneously, their processing order is unknown. For example, in Fig. 1, operations 1.1 and 1.2 (*createFile* and *readFile*) are issued almost simultaneously. We cannot know which operation is scheduled first. If operation 1.2 (*readFile*) is scheduled first, then an error "file not exist" will be thrown since the file 'a.txt' has not been created yet.

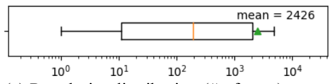
Non-deterministic event triggering. For two asynchronous operations scheduled by the worker pool, their completion order is unknown. The system environments, e.g., CPU, can affect the completion order of the operations. For example, in Fig. 1, if operation 2.2 completes before operation 2.1, the *read done* event of operation 2.2 will be put into the queue first. Thus, callback cb_2 will be invoked before cb_1. Besides, a network request may arrive at any time, its order relative to other events, e.g., completion events of asynchronous operations, is unknown.

Non-deterministic event handling. It is possible to have multiple callbacks available for execution at any point, and the choice for which one of these callbacks to schedule next is non-deterministic (as discussed earlier in Section II.A). Thus, callbacks may not be processed in their expected order.

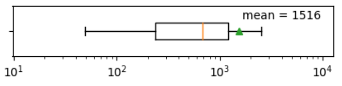
Non-determinism among multiple Node.js processes. The looper thread in Node.js is single-threaded. So, in order to take advantage of multi-core processors, Node.js can spawn many processes to distribute the workload. Node.js provides an elegant solution, *cluster* [17], to scale up applications by splitting a single process into multiple ones. These Node.js processes may have conflicting accesses to the same resources, e.g., a Node.js process copies a file in a directory while another process deletes the directory. This non-determinism could introduce concurrency bugs.

III. METHODOLOGY

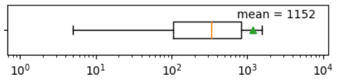
To answer the research questions RQ1-4, we collect real world concurrency bugs in open-source Node.js applications as our study subjects. This section presents how we collect real world concurrency bugs from Node.js applications, and further explains the methodology of our characteristic study.



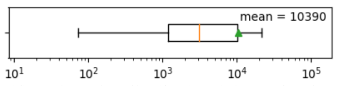
(a) Popularity distribution (# of stars)



(b) Code revision distribution (# of revisions)



(c) Issue distribution (# of issues)



(d) Project size distribution (Lines of code)

Fig. 2. Statistics of our studied Node.js projects.

A. *Collecting Concurrency Bugs*

Existing bug studies, e.g., [6][7], often first identify the mature projects, and then collect bugs in the projects. However, Node.js is relatively new, and its applications are still young. We find that Node.js applications usually report very few concurrency bugs. Therefore, in order to collect sufficient bugs for our study, we directly collect concurrency bugs from all Node.js projects in GitHub. Our approach to collect concurrency bugs in Node.js is described as follows.

Step 1: Searching concurrency bug reports in GitHub Node.js projects. As the most popular open-source platform, GitHub has hosted about 97,000 Node.js projects. Through an initial investigation on some Node.js projects, we find that there is no clear categorization for concurrency bugs in these projects. Thus, we use concurrency-related keywords to search candidate concurrency bugs in Node.js projects from GitHub. The keywords we used are as follows: "concurrent", "race", "synchronization", "atomic", "mutex", "transaction", "deadlock", "compete" and "starve". To answer our research questions, we further use advanced search conditions provided by GitHub to filter out bugs that are labeled as *bug*, already in *closed* state, and contains keywords like "submit" or "fix". This implies that these bugs are confirmed as bugs and may have fixing solutions. After this step, we obtain 1,583 bug reports.

Step 2: Selecting concurrency bugs in Node.js projects. Since containing the previous mentioned keywords does not mean that a selected bug report really contains a concurrency bug in Node.js, we manually validate these 1,583 bug reports, and exclude bug reports that are not related to concurrency and Node.js. Finally, we obtain 214 concurrency bug reports in 147 Node.js projects. We further carefully inspect these 214 concurrency bug reports, and only keep the concurrency bug reports that contain enough information to answer the research questions RQ1-4. Some concurrency bug reports are incomplete, e.g., we cannot find its corresponding bug fixing patch, or the bug report only contains a one-line description and we cannot infer how the bug happens. We exclude these incomplete bug reports from our characteristic study. After this step, we obtain 57 concurrency bugs for our bug characteristic study.

These 57 concurrency bugs come from 53 Node.js projects, including 12 server-side applications, 6 desktop applications, and 35 libraries, e.g., network communication API socket.io [18]. Fig. 2 shows the statistics of our studied 53 projects. These projects are *popular*: there are on average 2,426 stars, and 32% of them have more than 1,000 stars (Fig. 2a). These projects are *well-maintained*: there are on average 1,516 revisions and 1,152 issues; 56% of them have more than 500 revisions (Fig. 2b), and 23% of them have more than 1,000 issues (Fig. 2c). These projects are *big* and *complicated*: there are on average 10,390 lines of JavaScript code, and 40% of them contain more than 5,000 lines of code (Fig. 2d).

B. Analyzing Concurrency Bugs in Node.js

We manually study all the 57 concurrency bugs to answer the research questions RQ1-4. Specially, we study the bug reports, bug fixing patches and related discussions (e.g., comments), and then assign them into different categories according to their bug patterns and root causes (Section IV), failure impacts (Section V), bug manifestation (Section VI) and fix strategies (Section VII), respectively.

Specially, we categorize concurrency bugs into different categories in three phases. (1) We propose initial categories in advance according to the taxonomy in related work [6][7]. (2) We carefully study each bug and try to assign it into different categories. If a bug does not belong to any known category, we create a new category for it. (3) In the last, we review the above categorization, and ensure that the categorization demonstrates common bug characteristics, does not overlap, and covers most bugs. In this phase, we may need to adjust current categories, e.g., merge categories, rename category names or remove useless categories. In the above process, we make sure that each bug is studied at least by two authors. If we have conflicts about the bug or categorization, we will reinvestigate the bug further, until we form a final decision.

C. Threats to Validity

Similar to other bug characteristic studies, our study is subject to the validity problem. Potential threats to the validity include the representativeness of our studied concurrency bugs and our study methodology.

Representativeness of the studied concurrency bugs. All concurrency bugs we studied are collected from open-source Node.js projects. The keywords we used to select these bugs are a union set of keywords used by related studies [7][19][20]. We keep all the bugs that have enough information for categorization without bias. Further, our studied 53 projects include various types of Node.js applications, e.g., server-side applications and libraries. Most of these projects have high popularity and are well-maintained. Of course, some concurrency bugs may never be reported or fixed by developers. However, there is no proper way to study these bugs. We believe our studied bugs provide a representative sample of real world concurrency bugs in Node.js.

Study methodology. For each bug, every piece of information related to the bug, including developers' explanations, source code, fixing patches, and bug triggering test cases, needs to be studied. To minimize subjective errors in our study, each bug is studied by at least two authors in this paper.

IV. BUG PATTERNS AND ROOT CAUSES

Concurrency bugs in Node.js applications can have complex timing, involving multiple events and asynchronous operations. First, we study the triggering conditions to categorize the studied bugs into different bug patterns (Section A). Second, we study the root cause of each bug from two aspects, bug-inducing phase and bug-inducing API, to understand why these bugs are introduced by developers (Section B).

```
1.  var bundle = client.bundle('someBundle');
2.  bundle.create(…, function(){         // create is async
3.+   bundle.uploadStr(…);
4.  });
5.- bundle.uploadStr(…);                 // uploadStr is async
```

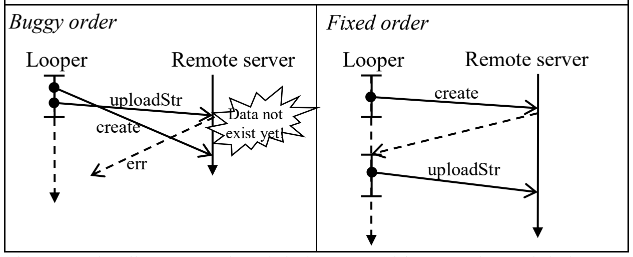


Fig. 3. gp-js-client#4: Order violation caused by non-deterministic execution of asynchronous operations [21]. *The code snippet aims to create a data in the remote server in line 2 and then updates the data in line 5 (create and uploadStr are two asynchronous operations). The two asynchronous operations are issued almost simultaneously, and may result in a possible buggy order in which the uploadStr operation is first processed by the remote server and returns an error "the data to update does not exist". The fix is to move the uploadStr operation into the callback of the create operation, so that the two asynchronous operations are ordered, and can be executed sequentially.*

A. Bug Patterns

We study the bug reports of the selected 57 concurrency bugs in Node.js, and then assign them into different categories according to their triggering conditions. Finally, we categorize these 57 bugs into three bug patterns: order violation, atomicity violation and starvation. Note that, we do not find deadlock bugs in Node.js, which are common in multi-threaded systems [6].

Order violation. About one third (17/57=30%) of the studied concurrency bugs are caused by violation to developers' order intention between two events or asynchronous operations. In this case, two or more events / asynchronous operations, which access the same shared resources (e.g., variables, files), are expected to be processed in a certain order. However, the order is not enforced during execution.

Order violation can happen in two situations. First, it happens between two asynchronous operations. Consider bug gp-js-client#4[1] [21] in Fig. 3. The two asynchronous operations *create* and *uploadStr* are issued one after another. Developers assume that these two operations are executed sequentially. Since these two operations are asynchronous, Node.js may execute them in any order. In the buggy order, the *uploadStr* operation is first processed by the remote server and returns an error "the data to update does not exist". To avoid this buggy order, the *uploadStr* operation can be moved to the callback of the *create* operation (Line 3). Second, order violation can also happen between two callbacks. In Fig. 1, the callbacks of two operations (steps 1.1 and 1.2), i.e., cb_1 and cb_2, are expected to be invoked as $cb_1 \rightarrow cb_2$. However, this order is not enforced by user code and there is a chance that cb_2 is invoked before cb_1.

Atomicity violation. About two thirds (37/57=65%) of the studied concurrency bugs are caused by violation to developers' atomic intention among several callbacks or asynchronous

[1]gp-js-client#4 denotes the concurrency bug reported by the issue 4 in project gp-js-client. Other concurrency bugs have the same representation in this paper.

```
1.  function addIdToArray (ownerName, id) {
2. -    return User.findOne({name: ownerName}).then(user => {
3. -      user._ids.push(id);
4. -      return user.save();
5. -    });
6. +  db.user.update({name:ownerName},{$push:{_ids: id}})
7.  }
```

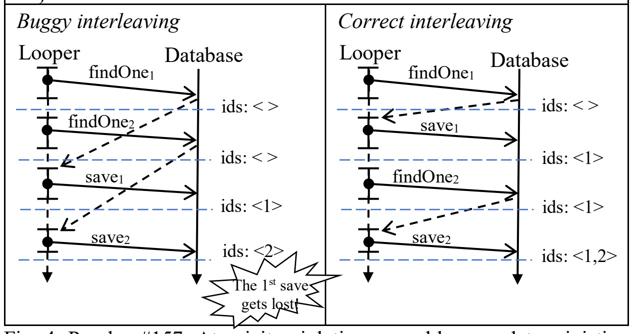


Fig. 4. Porybox#157: Atomicity violation caused by non-deterministic event triggering [22]. *Each user request (addIdToArray) triggers two asynchronous operations (findOne and save) in order. First, it triggers findOne operation (Line 2), which queries a list of IDs from the database asynchronously. Second, after findOne returns, it modifies the data and triggers save operation (Line 4), which saves the data back to the database asynchronously. For each user request, the above two steps should be atomic. However, two successive user requests with the same ownerName may cause buggy interleaving findOne$_1$ →findOne$_2$→save$_1$→save$_2$, resulting in the first save to the database getting lost (i.e., the IDs should be <1, 2>, not <2>). This bug is fixed by changing findOne and save into an atomic API (update) supported by MongoDB.*

```
1.   internals.emit = function (emitter, notification) {
2.     if (notification) {
3.       emitter._notificationsQueue.push(notification);
4.     }
5.     if (emitter.isProcessing || !emitter._notificationsQueue.length) {
6.       return;
7.     }
8.     emitter._isProcessing = true;
9.     const item = emitter._notificationsQueue.shift();
10.    const finalize = () => {
11.      if (item.callback) {
12. –      setImmediate(() => item.callback());
13. +      process.nextTick(itemCallback, item);
14.      }
15.      emitter._isProcessing = false;
16. –    setImmediate(() => internals.emit(emitter));
17. +    process.nextTick(emitEmitter, emitter);
18.    };
19.    //Some code invoking finalize asynchronously.
20.  }
```

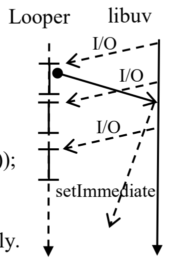


Fig. 5. hapi#3347: Starvation caused by non-deterministic event handling [24]. *An I/O request results in the invocation of internals.emit (Line 1) which pushes a notification to _notificationsQueue (Line 3). If no notification is in process, it then shifts an item from the queue (Line 9), processes it in a deferred setImmediate task (Line 12), and invokes internal.emit in another deferred task (Line 16) to process the remaining notifications in the queue later. This will cause a starvation bug since the event loop keeps processing the I/O event queue when the I/O is busy before it proceeds with the event queue that the setImmediate event lies in. As a result, the _notificationsQueue increases a lot before the notifications are processed and ends up throwing the error "allocation failed - process out of memory".*

operations. In this case, a set of events / asynchronous operations are assumed to be processed atomically without interruption, but the atomicity is not enforced during execution.

Consider bug Porybox#157 [22] in Fig. 4. When a user requests to add a new *id*, the function *addIdToArray* first triggers an asynchronous operation *findOne* (Line 2) to query the user's current *ids* from the database, and then triggers another asynchronous operation *save* (Line 4) to save the updated *ids* back to the database. Note that, for each user request, the above two asynchronous operations are correctly ordered through nested callbacks, and thus there is no order violation. For each user request, developers assume that this small code region will be executed atomically. However, the two operations *findOne* and *save* are executed in response to two different events, and other events may interleave in between. For example, if two user requests with the same username (*ownerName*) arrive simultaneously, it may trigger the following interleaving *findOne*$_1$→*findOne*$_2$→*save*$_1$→*save*$_2$. Here, the subscripts denote the user request ID. In this case, *findOne*$_1$ and *findOne*$_2$ both obtain the same *ids* (e.g., < >). For the first request, it updates *ids* from < > to <1>, and then saves *ids* to the database (*save*$_1$). For the second request, it updates *ids* from < > to <2>, and then saves *ids* to the database (*save*$_2$). Now, *ids* in the database becomes <2>, and the update by the first request is lost. The correct interleaving of two user requests should be *findOne*$_1$ →*save*$_1$ →*findOne*$_2$ →*save*$_2$, as shown in the right part of Fig. 4. In this case, the second request updates *ids* based on the first request's result. Thus, *ids* in the database becomes <1, 2>.

Atomicity violation can also originate from multiple processes. Two bugs are in this case. In bug cordova-lib#7 [23], the application (a tool for installing/uninstalling plugins) extracts each .tgz file into the same directory and then copies the extracted files out. When installing two different plugins: (1) the first process has extracted a .tgz file into the directory, and then is copying the extracted files; (2) the second process is extracting another .tgz file into the same directory. Thus, the first process can copy some files that belong to the second process.

Starvation. Few concurrency bugs (3/57=5%) happen when some tasks take a long time and prevent other events from processing. Usually, callbacks registered by higher priority can starve the lower ones.

Consider bug hapi#3347 [24] in Fig. 5. For each I/O request from the client, the *internals.emit* (Line 1) is called to push the *notification* to the queue *_notificationsQueue* (Line 3). If no notification is in process (Lines 5-7), the first notification is obtained from the queue *_notificationsQueue* (Line 9) and scheduled in a deferred task through *setImmediate* (Line 12). It further iteratively emits the *internals.emit* in another deferred task in order to process the remaining notifications in the queue *_notificationsQueue* (Lines 16). As mentioned in Section II.A, the I/O events have higher priority than events scheduled by *setImmediate*. So, if the I/O keeps busy, the events scheduled by *setImmediate* will have no chance to be processed. As a result, the *_notificationsQueue* keeps increasing and hits an error "allocation failed - process out of memory".

Finding #1. Concurrency bugs in Node.js can be categorized into three simple bug patterns: order violation, atomicity violation, and starvation. Two thirds of the studied bugs are atomicity violation.

TABLE 1. BUG-INDUCING PHASE

	Order	Atomicity	Starvation	Total
Asynchronous operation	4	16 (9)*	0	20 (9)*
Event triggering	12	28 (9)*	0	40 (9)*
Event handling	0	0	3	3
Multi processes	1	2	0	3

**9 atomicity violation bugs, shown in ( ), can be introduced by both non-deterministic execution of asynchronous operations and event triggering.*

B. Root Causes

It is difficult to know why these concurrency bugs were introduced by developers. By analyzing bug patterns and related bug descriptions in the bug reports, we try to understand the root causes from the following two aspects. (1) Bug-inducing phase: in which phase non-determinism is introduced? (2) Bug-inducing API: what asynchronous APIs are responsible for the racing events? Further, we try to postulate some possible and common misunderstanding behind these concurrency bugs.

Bug-inducing phase. As discussed in Section II.B, concurrency bugs can be introduced in three different phases: execution of asynchronous operations, event triggering and event handling. As shown in Table 1, concurrency bugs mostly happen due to non-determinism in event triggering (40/57=70%) and execution of asynchronous operations (20/57=35%). Only 3 bugs happen due to non-deterministic event handling. Note that 9 bugs are introduced by both non-deterministic execution of asynchronous operations and event triggering, so the accumulated percentages exceed 100%. Additionally, 3 bugs happen among multiple Node.js processes.

Bug gp-js-client#4 in Fig. 3 is caused by non-deterministic execution of asynchronous operations. In this example, the two asynchronous I/O operations (i.e., *create* and *uploadStr*) may be reordered by the underlying worker pool. Fig. 4 shows an example of concurrency bug due to non-deterministic execution of asynchronous operations and event triggering. For the buggy interleaving, the Node.js framework determines the two asynchronous I/O operations (i.e., *findOne* and *save*) in one request happen in order. However, the event order between two requests is non-deterministic. While, Fig. 5 shows an example of concurrency bugs due to non-deterministic event handling. This kind of bugs are rare since they only relate to non-deterministic event schedule for the looper thread.

Bug-inducing API. Understanding which asynchronous APIs are responsible for the racing events (i.e., events involved in a concurrency bug) is important for understanding the root cause of a concurrency bug. We categorize related APIs into two categories: *Schedule API* and *high-level API protocol*. Schedule APIs refers to native APIs provided by Node.js, e.g., *setTimeout*, *process.nextTick* and *Promise* [25], which are used to schedule deferred tasks. The APIs in Node.js are usually developed in the asynchronous and event-drive style. High-level API protocol refers to the asynchronous and event-driven specification that developers should follow when they use those APIs. The code snippet in Fig. 3 shows a typical example for high-level API protocol: for a *bundle*, before it is created by the *create* asynchronous operation, no further actions on the bundle (e.g., *uploadStr*) should be performed. For another example, a *xlsx* file extraction API provides two types of events: *row* event issued when a row in the *xlsx* file is parsed, and *end* event issued when the whole file is parsed. The *end* event should be processed only after all *row* events have been processed. Bug xlsx-extract#7 [26] does not respect this protocol, and causes an exception.

TABLE 2. BUG-INDUCING API

	Cases	Order	Atomicity	Starvation	Total
Schedule API	setTimeout	2	5	0	7
	process.nextTick	0	0	2	2
	setInterval	1	0	0	1
	setImmediate	0	0	1	1
	Promise	0	0	0	0
API protocol		11	17	0	28

Table 2 shows the statistics about bug-inducing APIs. Note that, the remaining bugs are not related to schedule APIs or wrong assumptions about high-level API protocols. For example, two user requests trigger two events that modify the same variables non-deterministically. We exclude them in Table 2.

- *Schedule API in Node.js*. Asynchronous operations can be scheduled by the commonly used schedule APIs. 11 (19%) concurrency bugs schedule their events with schedule APIs in Node.js, including *setTimeout* (7), *process.nextTick* (2), *setInterval* (1), *setImmediate* (1), and *Promise* (0). Thus, these schedule APIs can introduce non-determinism to Node.js applications.
- *High-level API protocol*. 28 (49%) concurrency bugs are caused by improper high-level API usage, e.g., bugs gp-js-client#4 and xlsx-extract#7 [26]. The asynchronous or event-driven-style protocols are usually not clearly described or understood by developers. Thus, developers may have wrong assumptions of the event order and/or atomicity. For example, in bug sequelize#1599 [27], a developer commented: "*I would expect findOrCreate() method to be atomic. However, it just calls find() and if unsuccessful, it will call create().*" In bug kue#154 [28], a developer commented: "*The redis client.subscribe is asynchronous but that is completely ignored*".

Finding #2. Non-deterministic event triggering (70%) and execution of asynchronous operations (35%) are two main sources of concurrency bugs, while existing work only focuses on non-deterministic event triggering. 28 (49%) bugs are caused by using high-level API protocols in an improper way.

V. BUG IMPACTS

We study the failure symptom of each bug to better understand how severe a concurrency bug is. Our studied concurrency bugs can cause fatal failures including crashes / exceptions, incorrect states, wrong outputs, hangs / no response, and other operation failures. Note that, incorrect states and wrong outputs can also cause some operation failures. To avoid double counting, we consider a bug as causing operation failures only when it does not cause incorrect states and wrong outputs.

Crashes / Exceptions. 19 (33%) of the studied bugs can cause crashes or uncaught exceptions, e.g., null pointer exception. For example, in bug hapi#3347 (Fig. 5), Node.js crashes due to out of memory.

TABLE 3. PRECONDITIONS

	Cases	Order	Atomicity	Starvation	Total
External request	0	13	8	2	23
	1	1	1	0	2
	2	2 (1)*	26 (17)*	0	28 (18)*
	>=3	1	2	1	4
Configuration		1	2	1	4
Deploy environment		0	1	1	2

**The bugs, shown in ( ), only require 2 same external requests with the same inputs.*

Incorrect states. 18 (32%) of the studied bugs can cause incorrect states, e.g., incorrect persistence data in database. For example, in bug Porybox#157 (Fig. 4), the first update is lost in the database.

Wrong outputs. 6 (11%) bugs can generate wrong results and present them to users.

Hangs / no response. 4 (7%) bugs can cause hangs or no response. For example, in bug fiware-pep-steelskin#269 [29], the application registers a listener for an event. However, an unexpected event may happen and remove the listener before it is triggered. As a result, the event cannot be processed correctly.

Operation failures. The other 10 (17%) bugs cause unexpected behaviors: jobs getting processed incompletely (2), jobs getting rejected (2), jobs getting processed twice (2), I/O starvation issues under heavy load (2), and others (2), e.g., a server cannot be shut down normally.

Finding #3. All the studied concurrency bugs in Node.js can cause severe consequences, e.g., crashes, incorrect states, hangs, and operation failures.

VI. BUG MANIFESTATION

Understanding how concurrency bugs manifest themselves in Node.js applications can provide useful implications on how to effectively detect and test concurrency bugs.

A. *Input Preconditions*

While Section IV.A presents the timing conditions of concurrency bugs, this section focuses on input preconditions. In practice, many input conditions should be satisfied in order to trigger the concurrency bugs, such as external requests, application configuration, and deploy environment. For example, in bug Porybox#157, two external requests with the same *ownerName* are required. Otherwise, this concurrency bug cannot be triggered. Table 3 shows the input preconditions for our studied bugs.

External requests. Node.js is usually used to process user requests in server-side applications. These user requests are external to Node.js applications, and can happen anytime. As Table 3 shows, 23 (40%) bugs do not need any external request. These bugs usually occur in desktop applications and libraries, and they do not receive any external request. 2 bugs only need 1 external request. 28 (49%) bugs need only 2 concurrent requests, and interestingly, 18 of them only require 2 *same* requests with the *same* inputs (e.g., Porybox#157). For the remaining 10 bugs, they require 2 different external requests. Only 4 concurrency bugs require more than 2 external requests.

We observe that about three quarters (43/57=75%) of the studied bugs can manifest themselves with no more than one external request or two same requests. This indicates that input preconditions are usually simple, and developers can test their applications with simple input preconditions first. The other 14 bugs need relatively complicated conditions and are not easy to trigger. They require at least two different external requests and may also require a specific order among them. For example, bug browser-laptop#3273 [30] is triggered only when three external requests are ordered as *on → off → on*.

Configuration and deploy environment. We observe that 6 (11%) bugs require special configuration (4 cases) or deploy environment (2 cases). For example, it requires specific database for bug five-bells-shared#64 [31] and certain Node.js version for bug asset-smasher#8 [32] to be manifested.

Finding #4. Three quarters of the studied concurrency bugs only require simple input preconditions. 11% of the studied bugs require special configuration or deploy environment.

B. *Racing Resources*

Racing resources usually denote the states of an application and a concurrency bug is introduced when these states hold unexpected or inconsistent values. We categorize racing resources as shared variable, database, file, and others.

Shared variable. 31 (54%) bugs contend against shared variables. Shared variables are commonly used to store shared data in memory, or as condition variables used for synchronization between callbacks.

Database. 15 (26%) bugs contend against the data in database. This usually happens when several database operations (e.g., query and update) are not ordered or executed in an atomic region, and cause inconsistency in the database. For example, bug Porybox#157 in Fig. 4 contends against database.

File. 8 (14%) bugs contend against files, e.g., bug cordova-lib#7 [23]. This usually happens when several file operations (e.g., read and write) to the same file are not ordered.

Other. 3 bugs do not have racing resources, such as a starvation bug that is only sensitive to event schedule.

Finding #5. A significant number (23 out of 57) of concurrency bugs contend against databases or files, rather than variables.

C. *Triggering Scope*

We analyze the triggering scope to provide a complexity measure of the triggering of a concurrency bug in Node.js. We use the following metrics to measure the triggering scope: events / asynchronous operations count, Node.js process count, and racing resource count, as shown in Table 4.

Events / asynchronous operations. For each concurrency bug, we identify the smallest set of events / asynchronous operations *E*, so that a specific order of *E* can guarantee that the bug manifests. As Table 4 shows, most (93%) bugs only involve no more than 4 events / asynchronous operations. This indicates that testing Node.js applications and detecting concurrency bugs in Node.js can be simplified to check no more 4 events / asynchronous operations without losing bug detection capability much.

TABLE 4. TRIGGERING SCOPE

	Cases	Order	Atomicity	Starvation	Total
Events / asynchronous operations	2	15	6	0	21
	3	0	14	0	14
	4	2	16	0	18
	>4	0	1	3	4
Involved processes	1	16	35	3	54
	2	1	2	0	3
Racing resources	0	1	0	2	3
	1	14	34	1	49
	2	2	2	0	4
	3	0	1	0	1

Involved Node.js processes. 54 (95%) concurrency bugs only involve 1 Node.js process. This indicates most concurrency bugs happen in one single Node.js process and bug detection approaches only focusing on one Node.js process can cover most concurrency bugs.

Racing resources. 49 (86%) bugs race for only 1 resource (e.g., a shared variable, a file or an entity in database). Bug change-propagation#84 [33] is an example of racing for two shared variables: the program makes a commit to database if certain conditions about two shared variable are satisfied.

Finding #6. Most concurrency bugs only involve no more than 4 events / asynchronous operations, 1 Node.js process, and 1 resource.

VII. BUG FIXING

A. Fix Strategies

Before we investigate how concurrency bugs in Node.js were fixed in practice, our intuition is that adding synchronization (forcing orders among events / asynchronous operations) should be the most common way to fix concurrency bugs in Node.js. Surprisingly, it is not the case. In total, we summarize 8 fix strategies that can fix 46 out of 57 bugs, as shown in Table 5. Only one quarter of the bugs were fixed by adding synchronization. There are several potential reasons. First, it is difficult to enforce atomic intentions by forcing orders among callbacks or asynchronous operations. Second, simply using synchronization on all the operations can lower the parallelization and thus degrade the performance. Therefore, about three quarters of bugs were fixed without synchronization. Developers usually need to consider correctness and performance to decide the most appropriate fix strategies. In the following, we describe these different strategies.

Adding synchronization. One quarter of concurrency bugs were fixed by changing the timing of callbacks or asynchronous operations. Node.js can use callbacks or equivalent async control flow libraries, e.g., async[34] and syncify[35], to add synchronization. For example, bug gp-js-client#4 in Fig. 3 was fixed by moving *uploadStr* into the callback of *create* (Line 3). Thus, the order of these two asynchronous operations is enforced. Node.js can also use condition variables to enforce orders. For example, in the code snippet in Fig. 5, two callbacks (Lines 13 and 17) by *process.nextTick()* should be invoked as an atomic region. It uses a condition variable *isProcessing* to avoid concurrent processing of external requests. If *isProcessing* is true, it just puts the new incoming request in a queue (Line 3) and will reschedule it later. An equivalent way to do this is using

TABLE 5. FIX STRATEGIES

Fix strategies	Order	Atomicity	Starvation	Total
Adding synchronization	7	6		13
Bypassing	5	10		15
Tolerance	1	4		5
Switching to atomic APIs		4		4
Ignoring/retrying	1	1		2
Moving code		2		2
Data privatization		2		2
Changing priority			3	3
Other	3	8		11
Total	17	37	3	57

third-party lock [36] or mutex [37]. Only 7 out of 37 atomicity violation bugs were fixed by adding synchronization: 3 bugs were fixed by using shared variables, and 4 bugs were fixed by using third-party lock/mutex. Most atomicity violation bug were ***not*** fixed by adding synchronization. Developers would try to fix these bugs by tolerating wrong event timing instead of preventing the buggy timing with synchronization.

Bypassing. 15 bugs were fixed by bypassing code when certain variable conditions are satisfied. 5 order violation bugs were fixed this way. For example, bug js-ipfs#318 [38] results in calling a callback twice and finally throws an exception. It was fixed by introducing a variable denoting whether the callback has been called to avoid calling it again. 10 atomicity violation bugs were fixed by skipping code to avoid erroneous behaviors when its concerned variable is modified by an unexpected event (i.e., under buggy interleaving). The fix of bug strider#745 [39] just moves on if certain variable is not null.

Tolerance. 5 bugs were fixed by tolerating the buggy event timing. 1 order violation bug was fixed by tolerating buggy timing. For example, bug session#340 [40] was fixed by correcting the states of shared variables so that the following code can also run under buggy event order. 4 atomicity violation bugs were fixed by tolerating the incorrectly interleaved events. For example, the fix of bug fiware-pep-steelskin#279 [41] corrects the value of the concerned variable when it detects the variable is modified by an unexpected event's callback. Note that, this kind of fixes are usually semantic-related, developers need to confirm how to update the corrupted states.

Switching to atomic APIs. 4 bugs were fixed by replacing current APIs with their atomic versions. For example, bug Porybox#157(Fig. 4) was fixed by replacing *findOne* and *save* with the atomic API *update* which can do the same task in a single query supported by MongoDB.

Ignoring / retrying. 2 bugs were fixed by catching the failure or retrying the failed operation. For example, bug browser-laptop#3273 was fixed by catching the error and showing it to users. In bug done-ssr#62 [42], the application launches the component *can-serve* and *live-reload* at the same time, the *can-serve* may get ready first and tries to send a request to the *live-reload* server which has not started yet. This bug was fixed by letting the *can-serve* retry the request later.

Moving code. 2 atomicity violation bugs were fixed by moving code. Their fixes merge the supposed atomic operations but separated in two callbacks together in one callback. Thus, they are guaranteed to be executed atomically.

TABLE 6. FIX COMPLEXITY

	25th percentile	Median	75th percentile	Max
Time (# of days)	2	6	31	832
# of comments	2	5	10	49
LOC of final patch	5	13	28	250
# of patches	1	2	3	6

Data privatization. 2 bugs were fixed by making shared variables private under the same buggy context. Thus, racing events' callbacks cannot access the shared variables. 2 atomicity violation bugs were fixed by data privatization.

Changing priority. 3 starvation bugs were fixed by adjusting the priorities of relevant events. The 7 event queues have different strategies, as shown in Section II.A. For example, recursively scheduling events by *process.nextTick* can starve the event loop since it has very high priority. This can be fixed by using *setImmediate* that has relatively lower priority. Similarly, I/O events can starve callbacks registered by *setImmediate*. This can be fixed by using *process.nextTick* that has higher priority.

Other. The remaining 11 bugs were fixed by various ad-hoc approaches, such as, changing underlying C/C++ Node.js plugin code, updating dependent databases, and redesigning related data structure and code logic.

Finding #7. Most (81%) of the studied bugs can be fixed by a small set of fix strategies. Three quarters of the studied bugs are not fixed by simply adding synchronization.

B. Fix Complexity

To quantify the effort and complexity of fixing the concurrency bugs in Node.js, we use five metrics to measure the fix complexity: (1) the time to resolve the bug, (2) the number of bug comments among developers, (3) the patch size in terms of LOC changed, (4) the number of patches submitted, (5) the number of shared variables introduced during fixing.

For the first four metrics, we extract the corresponding information from the bug reports in GitHub. Table 6 shows the statistics of the first four metrics. On average, our studied bugs take 55 days to fix, have 8 comments, submit 2 patches and have 29 lines of code changed. Due to space constraints, we do not provide further cross-cutting analyses, e.g., how many bugs have patches with more than 50 LOC.

Shared variables introduced in bug fixes. Due to lack of synchronous primitives, e.g., *lock* and *wait/notify*, shared variables are prevalently used for fixing concurrency bugs in Node.js. Here, we only count the number of shared variables, which are used as condition variables for checking program states, or to store state information for bypassing or tolerance. The use of shared variables can complicate fixes since developers need to consider how to form the conditional statement and where to place it.

We analyze how many shared variables are introduced in the patches. For 27 concurrency bugs, shared variables are used as condition variables or to store state information. For these bugs, 17 bugs use 1 shared variable, 7 bugs use 2 shared variables, 2 bugs use 3 shared variables and 1 bug uses 4 shared variables. For bug fixing, 17 bugs introduce new shared variables to fix bugs. Among these 17 bugs, 8 bugs have used shared variables for preventing concurrency bugs, but they were used in the wrong way. The final patches fix them. This also indicates fixing concurrency bugs with shared variables is difficult.

Finding #8. Using shared variables to fix bugs are more often and error-prone: In a half of the studied bugs, they use shared variables, as condition variables or state check/recovery, to prevent concurrency bugs. One third of the fixes introduce new shared variables.

VIII. LESSONS LEARNED

We now discuss the lessons learned, implications to existing tools and the opportunities for new research in combating concurrency bugs in Node.js.

A. Concurrency Bug Detection in Node.js

Concurrency bugs in Node.js can cause severe consequences (Finding #3), and thus resolving these bugs is of great significance for the reliability of Node.js applications. Since Node.js is new, concurrency bug detection tools are unfortunately rare. Our study provides some patterns and guidance that can facilitate future studies on concurrency bug detection in Node.js.

Pattern-based bug detection. Finding #1 implies that concurrency bug detection in Node.js can focus on three simple bug patterns: order violation, atomicity violation and starvation. Although there exists some concurrency bug detection tools in other event-based systems, like Android[10][11] and client-side JavaScript [8][12], they mainly focus on order violation. Atomicity violation bugs are not well addressed in event-based systems yet. However, atomicity violation bugs are dominant (Finding #1). Thus, new bug detection approaches should be developed to address atomicity violation in Node.js.

Resource-oriented bug detection. Finding #5 shows that a significant number (23 out of 57) of concurrency bugs contend against databases or files, other than shared variables. While existing concurrency bug detection tools mostly focus on shared variables (i.e., shared memory). New approaches to detect concurrency bugs on these shared resources, e.g., databases and files, are needed. Researchers may build the access models on these shared resources to facilitate concurrency bug detection.

API-usage-guided bug detection. The asynchronous or event-driven-style protocols in Node.js APIs may not be clearly described, and then incorrectly understood by developers. Finding #2 provides empirical evidence that automatically extracting protocols and checking API uses against the protocols can be effective to detect concurrency bugs in Node.js.

System testing. Testing plays an important role in exposing concurrency bugs. However, few testing techniques are proposed for Node.js [43]. It is challenging to systematically test all possible event interleavings for Node.js applications. Our Finding #4 implies that, to reduce the test complexity, developers can focus on testing applications with simple input preconditions for triggering most concurrency bugs. However, testing also needs to take configurations and deploy environments into consideration (Finding #4). Furthermore, Finding #6 implies that testing can be simplified to check no more than 4 events in 1 Node.js process, without losing bug detection capability much. This will be more effective than testing all possible interleavings.

B. *Bug Fixing in Node.js*

Recent studies on concurrency bug fixing mostly focus on multi-threaded programs, and fix bugs by inserting lock/unlock [44][45]. Due to lack of lock mechanism in Node.js, these approaches cannot be directly used for concurrency bugs in Node.js. ARROW [14] and EventRaceCommander [46] fix order violation bugs using ad-hoc synchronization for client-side web applications. They need to take advantage of UI features (e.g., DOM) to build precise happens-before model that Node.js does not have. Further, Finding #7 shows that using synchronization is not appropriate for most (77%) bugs in Node.js. Future studies should consider other fix strategies (e.g., bypassing or tolerance) and try to generate high-quality fixes that are similar to human-written ones.

C. *API Design and Comprehension in Node.js*

In Node.js, APIs often are written in asynchronous and event-driven style, while developers may consider them as synchronous and non-event-driven APIs. Finding #2 indicates that developers can easily make wrong assumptions about how the APIs are used, and thus introduce concurrency bugs. This indicates that good API specifications on their asynchronous protocols should be helpful to avoid concurrency bugs. This also presents a unique opportunity for developing and inferring good API usage patterns in Node.js.

D. *Transaction Support in Node.js*

Finding #1 shows two thirds of our studied bugs are atomicity violation bugs. However, no convenient way exists to express developers' atomic intentions in Node.js (i.e., two or more events should be processed without interruption). In Section VII.A, we can see developers use some strategies to fix concurrency bugs, e.g., preventing other events from atomic regions, atomic API, consistency check and retry. It is not easy to adopt these fix strategies, as they usually introduce condition variables, state variables for recovery, and so on. However, these strategies are similar to transactional memory (TM) [47]. Based on our initial analysis about whether TM can help avoid the studied concurrency bugs, we find that 31 (54%) bugs can benefit from TM. Thus, TM can be treated as an efficient approach to avoid concurrency bugs in Node.js. More studies are needed to design a simple and effective TM in Node.js.

IX. Related Work

Our study relates to a large body of existing work on detecting, debugging, and understanding for concurrency bugs. In this section, we discuss some representative work in bug studies, concurrency bug analysis, and program analysis.

Bug studies. There are some representative works on bug studies in JavaScript and concurrency bugs. (1) Existing bug studies in JavaScript mainly focus on client-side JavaScript applications [48][49][50]. Many bugs in these studies relate to DOM, whereas Node.js does not have DOM. These studies merely mention concurrency bugs. Only the study [50] observes that non-deterministic errors are common in web applications. Node.fz [43] provides an initial study on a small set of concurrency bugs (only 12 bugs) in Node.js, to help design a fuzzing tool for Node.js. Whereas, we perform a compressive study on 57 concurrency bugs in Node.js, and obtain many new findings and implications, e.g., API misuse-related concurrency bugs, many new fix strategies, various new statistics and lessons learned. (2) Some studies have focused on concurrency bugs in multi-threaded systems [6] and distributed systems [7]. These studies have promoted a large amount of research on concurrency bug detection, testing, automated fixing and so on [9][51][52][53]. However, the concurrency bugs in Node.js differ from those in traditional systems as they originate from different programming paradigms and execution environments.

Concurrency bug analysis. Amounts of studies focus on concurrency bug detection [54][55][56], testing [9][57][58][59], reproduction [60] and fixing [46][44] in traditional programs (e.g., C and Java). In recent years, much research effort has also been devoted to concurrency bugs in event-driven applications, e.g., Android [10][11][13][61] and Web applications [8][12][14][62]. Our study shows that concurrency bugs in Node.js have different characteristics in bug patterns, manifestations and fix strategies from theirs. Thus, those approaches may be ineffective for Node.js. Our comprehensive study on concurrency bugs in Node.js provides further motivations and guidance for future studies.

Program analysis on Node.js applications. Madsen et al. [63] build an event-based call graph for a Node.js application, and then use it to statically detect bugs related to event handling, e.g., dead event listeners. Their tool is not designed for finding and resolving concurrency bugs in Node.js. SYNODE [64] combines static analysis and runtime enforcement of security policies to allow vulnerable modules in Node.js to be used in a safe way. SAHAND [65] builds asynchronous interactions for full-stack JavaScript applications. It captures the behavioral model of a full-stack JavaScript application, and provides scheduling lifelines of callbacks. Our study can open up new research directions on reliability issues in Node.js applications.

X. Conclusion

Node.js has become one of most popular platforms for building server-side applications. Applications built on Node.js face various non-determinism and may contain intricate concurrency bugs. This paper presents NodeCB, a comprehensive study on real world concurrency bugs in Node.js. We collect 57 real world concurrency bugs from various open-source Node.js applications. We examine their bug patterns, root causes, failure symptoms, manifestation, and fix strategies. Our study reveals many interesting findings, which can promote future concurrency bug detection, testing, and automated fixing in Node.js. In the future, we will design approaches and tools for detecting, avoiding, and fixing concurrency bugs in Node.js.

Acknowledgments

This work was supported by National Key Research and Development Plan (2016YFB1000803), National Natural Science Foundation of China (61672506, 61702490), Beijing Natural Science Foundation (4164104), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

[1] "Developer Survey Results 2016." [Online]. Available: http://stackoverflow.com/research/developer-survey-2016.

[2] "New Node.js Foundation Survey Reports New 'Full Stack' In Demand Among Enterprise Developers." [Online]. Available: https://nodejs.org/uk/blog/announcements/nodejs-foundation-survey/.

[3] "Chrome V8." [Online]. Available: https://developers.google.com/v8/.

[4] "npm repository." [Online]. Available: https://www.npmjs.com/.

[5] "State of the Union: npm." [Online]. Available: https://www.linux.com/news/event/Nodejs/2016/state-union-npm.

[6] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008, pp. 329–339.

[7] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC:A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2016, pp. 517–530.

[8] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race Detection for Web Applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 251–262.

[9] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "DCatch : Automatically Detecting Distributed Concurrency Bugs in Cloud Systems," in *Preceedings of International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2017, pp. 677–691.

[10] 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)*, 2015, pp. 332–348.

[11] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race Detection for Event-driven Mobile Applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[12] V. Raychev, M. Vechev, and M. Sridharan, "Effective Race Detection for Event-Driven Programs," in *Proceedings of the ACM SIGPLAN international conference on Object oriented programming systems languages & applications(OOPSLA)*, 2013, pp. 151–166.

[13] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically Verifying and Reproducing Event-Based Races in Android Apps," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2016, pp. 377–388.

[14] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster, "ARROW : Automated Repair of Races on Client-Side Web Pages," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2016, pp. 201–212.

[15] "libuv." [Online]. Available: https://github.com/libuv/libuv.

[16] "The Node.js Event Loop." [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/.

[17] "Cluster." [Online]. Available: https://nodejs.org/api/cluster.html.

[18] "socket.io." [Online]. Available: https://github.com/socketio/socket.io.

[19] G. Pinto, W. Torres, and F. Castor, "A Study on the Most Popular Questions about Concurrent Programming," in *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, 2015, pp. 39–46.

[20] M. Yu, Y.-S. Ma, and D.-H. Bae, "Characterizing Non-deadlock Concurrency Bug Fixes in Open-source Java Programs," in *Proceedings of the Annual ACM Symposium on Applied Computing(SAC)*, 2016, pp. 1534–1537.

[21] "gp-js-client/issue#4: document that calls are async." [Online]. Available: https://github.com/IBM-Bluemix/gp-js-client/issues/4.

[22] "porybox/issue#157: when two boxes are added simultaneously, one of the IDs get lost." [Online]. Available: https://github.com/porygonco/porybox/issues/157.

[23] "cordova-lib/pull#7: extract plugin .tgz files in unique directories." [Online]. Available: https://github.com/Icenium/cordova-lib/pull/7.

[24] "hapi/issue#3347: major performance issue with hapi.js 15.x." [Online]. Available: https://github.com/hapijs/hapi/issues/3347.

[25] "ECMAScript 6." [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/.

[26] "xlsx-extract/issue#7: Exception 'write after end.'" [Online]. Available: https://github.com/ffalt/xlsx-extract/issues/7.

[27] "sequelize/issue#1599: findOrCreate is not atomic." [Online]. Available: https://github.com/sequelize/sequelize/issues/1599.

[28] "kue/issues#154: Race between Job.prototype.save() and done()." [Online]. Available: https://github.com/Automattic/kue/issues/154.

[29] "fiware-pep-steelskin/issue#269: Race condition causes requests that will never be responsed-new race introduced in a fixed version." [Online]. Available: https://github.com/telefonicaid/fiware-pep-steelskin/issues/269.

[30] "browser-laptop/pull#3273: guard against race condition." [Online]. Available: https://github.com/brave/browser-laptop/pull/3273.

[31] "five-bells-shared/issue#64: retry failed DB operations caused by concurrent updates." [Online]. Available: https://github.com/interledgerjs/five-bells-shared/issues/64.

[32] "asset-smasher/issue#8: stop starving the event loop." [Online]. Available: https://github.com/jriecken/asset-smasher/issues/8.

[33] "change-propagation/pull#84:Fix race condition in commit logic." [Online]. Available: https://github.com/wikimedia/change-propagation/pull/84.

[34] "async." [Online]. Available: https://github.com/caolan/async.

[35] "syncify." [Online]. Available: https://github.com/aldonline/syncify.

[36] "lock." [Online]. Available: https://www.npmjs.com/package/lock.

[37] "mutex." [Online]. Available: https://www.npmjs.com/package/mutex.

[38] "js-ipfs/issue#318: Uncaught Error: no writecb in Transform class." [Online]. Available: https://github.com/ipfs/js-ipfs/issues/318.

[39] "strider/issue#745: Strider crashes on new jobs started from a Pull Request that has been rebased." [Online]. Available: https://github.com/Strider-CD/strider/issues/745.

[40] "session/issue#340: race condition between session.touch and session.save." [Online]. Available: https://github.com/expressjs/session/issues/340.

[41] "fiware-pep-steelskin/issue#279: when the PEP receive two or more simultaneous requests." [Online]. Available: https://github.com/telefonicaid/fiware-pep-steelskin/issues/279.

[42] "done-ssr/issue#62: race condition between can-serve and live-reload." [Online]. Available: https://github.com/donejs/done-ssr/issues/62.

[43] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the Server-Side Event-Driven Architecture," in *Proceedings of the European Conference on Computer Systems(EuroSys)*, 2017, pp. 145–160.

[44] G. Jin, W. Xhang, D. Deng, B. Liblit, and S. Lu, "Automated Concurrency-Bug Fixing," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation(OSDI)*, 2012, pp. 221–236.

[45] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware Fixing of Concurrency Bugs," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 318–329.

[46] C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen, "Repairing Event Race Errors by Controlling Nondeterminism," in *Preceedings of International Conference on Software Engineering (ICSE)*, 2017.

[47] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 64–78, 2003.

[48] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An Empirical Study of Client-side JavaScript Bugs," in *Preceedings of the International*

Symposium on Empirical Software Engineering and Measurement(ESEM), 2013, pp. 55–64.

[49] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “A Study of Causes and Consequences of Client-Side JavaScript Bugs,” *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 2, pp. 128–144, 2016.

[50] F. S. Ocariza, K. Pattabiraman, and B. Zorn, “JavaScript Errors in the Wild: An Empirical Study,” in *Proceedings of International Symposium on Software Reliability Engineering(ISSRE)*, 2011, pp. 100–109.

[51] T. Zhang, C. Jung, and D. Lee, “ProRace: Practical Data Race Detection for Production Use,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 149–162.

[52] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng, “AI: A Lightweight System for Tolerating Concurrency Bugs,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 330–340.

[53] S. Lu, S. Park, and Y. Zhou, “Detecting Concurrency Bugs from the Perspectives of Synchronization Intentions,” *IEEE Transactions on Parallel and Distributed Systems(TPDS)*, vol. 23, no. 6, pp. 1060–1072, 2012.

[54] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” in *Proceedings of the ACM symposium on Operating systems principles (SOSP)*, 1997, vol. 15, no. 4, pp. 27–37.

[55] J. Huang and C. Zhang, “Debugging Concurrent Software: Advances and Challenges,” *Journal of Computer Science and Technology (JCST)*, vol. 31, no. 5, pp. 861–868, 2016.

[56] D. Schonberg, “On-the-fly Detection of Access Anomalies,” in *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, 1989, pp. 285–297.

[57] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated Concolic Testing of Smartphone Apps,” in *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[58] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, “Using GUI Ripping for Automated Testing of Android Applications,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 258–261.

[59] C. S. Jensen, M. R. Prasad, and A. Møller, “Automated Testing with Targeted Event Sequence Generation,” in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2013, pp. 67–77.

[60] J. Huang and C. Zhang, “LEAN: Simplifying Concurrency Bug Reproduction via Replay-supported Execution Reduction,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA)*, 2012, pp. 451–466.

[61] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, “Detecting Event Anomalies in Event-based Systems,” in *Preceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, 2015, pp. 25–37.

[62] Y. Zheng, T. Bao, and X. Zhang, “Statically Locating Web Application Bugs Caused by Asynchronous Calls,” in *Proceedings of the international conference on World wide web(WWW)*, 2011, pp. 805–814.

[63] M. Madsen, F. Tip, and L. Ondřej, “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)*, 2015, pp. 505–519.

[64] C.-A. Staicu, M. Pradel, and B. Livshits, “Understanding and Automatically Preventing Injection Attacks on Node.js,” 2016.

[65] S. Alimadadi, A. Mesbah, and K. Pattabiraman, “Understanding Asynchronous Interactions in Full-Stack JavaScript,” in *Proceedings of the International Conference on Software Engineering(ICSE)*, 2016, pp. 1169–1180.