Semantics of Asynchronous JavaScript

Technical report, 2017-07-26

Matthew C. Loring
Google Inc
mattloring@google.com

Mark Marron
Microsoft Research
marron@microsoft.com

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

The Node.js runtime has become a major platform for developers building cloud, mobile, or IoT applications using JavaScript. Since the JavaScript language is single threaded, Node.js programs must make use of asynchronous callbacks and event loops managed by the runtime to ensure applications remain responsive. While conceptually simple, this programming model contains numerous subtleties and behaviors that are defined implicitly by the current Node.js implementation. This paper presents the first comprehensive formalization of the Node.js asynchronous execution model and defines a high-level notion of *async-contexts* to formalize fundamental relationships between asynchronous events in an application. These formalizations provide a foundation for the construction of static or dynamic program analysis tools, support the exploration of alternative Node.js event loop implementations, and provide a high-level conceptual framework for reasoning about relationships between the execution of asynchronous callbacks in a Node.js application.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords JavaScript, Asynchrony

1 Introduction

JavaScript is one of the most popular programming languages in use today and is rapidly expanding beyond its traditional role of client-side scripting. Node.js is a major platform for building JavaScript applications for the server, cloud, mobile, and IoT platforms. This rapid growth in popularity and usage has resulted in a growing demand from developers for information on how to best use the many asynchronous API's exposed by Node.js and for tools that can help develop [15], debug [4], or monitor [10, 24] the asynchronous behavior of their applications.

1.1 Semantics of Node Event Queues

A major challenge for research and tooling development for Node.js is the lack of a formal specification of the Node.js asynchronous execution model. This execution model involves multiple event queues, some implemented in the native C++ runtime, others in the Node.js standard library API bindings, and still others defined by the JavaScript *ES6 promise* language feature. These queues have different rules regarding when they are processed, how processing is interleaved, and how/where new events are added to each queue. These subtleties are often the difference between a responsive and scalable application and one that exhibits a critical failure.

```
var cb = function() { process.nextTick(cb); }
fs.write(process.stdout.fd, 'hi', function() {
  /*never printed*/
  fs.writeSync(process.stdout.fd, 'done');
});
cb();
```

Fig. 1. nextTick starvation of fs.write callback

```
var cb = function() { setImmediate(cb); };
fs.write(process.stdout.fd, 'hi', function() {
  /*printed in finite time*/
  fs.writeSync(process.stdout.fd, 'done');
});
cb();
```

Fig. 2. setImmediate scheduling setImmediate is fair

Consider the following pair of programs that differ in only the use of a single Node API – `process.nextTick(cb)` (in Figure 1) vs. `setImmediate(cb)` (in Figure 2):

The first line of Figure 1 defines a function that, when called, will register itself to be executed again "once the current turn of the event loop turn runs to completion" [18]. The second line initiates an asynchronous write to stdout which, when completed, will invoke the argument callback which synchronously prints 'done' to stdout. Finally, the function `cb` is invoked. At the end of this code block, Node will return to the event loop and begin dispatching from the various queues.

Based on the Node.js event loop implementation, the first callback dispatched will be `cb` registered by `nextTick` even though this was added to the event loops after `fs.write` callback. Further, the code will never print 'done' to the console. Each call to `nextTick` inserts the `cb` callback in a special `nextTick` queue which is drained before any other I/O events. This results in the starvation of any I/O tasks including the callback that contains the `fs.writeSync` call.

If, instead of using `nextTick`, we use `setImmediate` as shown in the code block in Figure 2, then we should see 'done'

printed in the first iteration of the event loop[1]. This different behavior is due to the fact that `setImmediate` places the callbacks in another special queue that is drained after some (but perhaps not all) pending I/O based callbacks have been executed. This interleaving ensures that both the I/O and timer based callback computations will make progress thus avoiding starvation.

As can be seen in this example, it is not possible to realistically model the semantics of a Node application using a single queue or even a single queue draining rule [15]. Thus, our first goal in this work is to construct a realistic formalization of asynchronous execution in a Node application.

1.2 Asynchronous Execution Context

During the execution of a program there may be many asynchronous callbacks simultaneously queued for execution. These callbacks may be associated with different logical tasks such as requests in the case of a web server. For many applications preserving the logical identity of a distinct *asynchronous execution chain* is critical. For example a developer debugging their application may want to see only logging output associated with handling a single http request of interest or to know both the *wall time* taken to respond to a request as well as the cycles spent specifically processing it (as opposed to being blocked on I/O or waiting for other work to complete).

Given the importance of understanding and tracking asynchronous execution context in Node.js applications the community has developed several frameworks for tracking asynchronous callbacks including Zones [26], Hooks [12], and Stacks [25]. Fundamentally, each of these systems is based on an informally defined relation which associates the point where an asynchronous callback is added to a worklist, and the point where the callback is dequeued and invoked.

Despite the importance of the concept of an *async-context* and the multitude of tools that provide some variation of this context there is currently no widespread consensus between tools about what constitutes a link in an asynchronous event chain. This situation is further exacerbated by a lack of formalization about what *async-context* is and if the same definition of context is sufficient for all applications or if there is, in fact, more than one useful definition of context. Thus, our second goal in this work is to formalize the notion(s) of *async-context* in a manner that is independent of a specific worklist implementation, such as the current Node implementation, and show how this context can be computed from the high-level language and API definitions.

In summary this paper makes the following contributions:

- We introduce λ_{async} which extends λ_{js} with promise semantics and microtask queue/event loops to enable the formalization of Node.js asynchronous execution semantics.
- Building on the λ_{async} semantics we define the concepts of *causal context* and *linking context* for reasoning about async-context in an application and present context propagation rules for computing them.
- We illustrate how these concepts and formalization support existing applications with race detection and debugging examples.
- We show how our formalization of event-loop semantics and asynchronous contexts enable further research into the design and development of a resource-aware priority scheduler for Node.js.

[1] Assuming console I/O completes immediately which is not always true but the callback will always be scheduled fairly with `setImmediate` once it completes.

2 Node.js Asynchronous Event Loop Semantics

In order to precisely define the semantics of asynchronous context, we must first provide an accurate model for the runtime behavior of the asynchronous primitives available in Node.js. To address this, we define λ_{async} which extends λ_{js} [11] with asynchronous primitives. In particular, we define both the asynchronous callbacks of Node.js and JavaScript promises uniformly in terms of *priority promises*. A priority promise has the core semantics of a JavaScript promise and is augmented with a priority value. For this article we also leave out many features of real JavaScript promises like exception handling and chaining which can be represented in terms of the primitives we model. Similarly, we do not discuss the new ES2017 `async`/`await` syntax as these operations can also be expressed in terms of regular promises.

2.1 Syntax

Figure 3 defines the extended syntax of λ_{async}. For conciseness, we only present the extra constructs added over λ_{js}.

Values v are either regular JavaScript values or a *priority promise*. The … stands for the definition in λ_{js} as described in [11], basically constants, functions, and records. The new priority promise is a triple (n, r, fs) with a priority level n, a value r that is either unres for an unresolved promise, or $\text{res}(v)$ for a resolved promise, and a list of pending callbacks fs.

Expression e are extended with three operations for working with promises:

- *Promise*() creates a new promise (of priority 0).
- $e_1.\mathit{resolve}(e_2)$ resolves promise e_1 to the value of e_2.
- $e_1.\mathit{then}(e_2)$ schedules e_2 for execution once promise e_1 is resolved.

This interface deviates from regular promises where the `resolve` method is (usually) hidden and where the constructor function for a promise takes a function as an argument: A JavaScript $\mathit{Promise}(f)$ creates a new promise that executes f asynchronously and when it returns a value, the promise resolves to that value. In our model this can be expressed as:

$v \in Val$	::=	...	
	|	(n, r, fs)	promise tuple
r	::=	unres	unresolved
	|	res(v)	resolved to v
fs	::=	$[f_1, ..., f_m]$	callbacks
n			priority levels
$e \in Exp$	::=	...	
	|	*Promise*()	create a promise
	|	e.*resolve*(e)	resolve a promise
	|	e.*then*(e)	add a listener
	|	•	the event loop

Fig. 3. Syntax of λ_{async}.

```
function Promise(f) {
  var p = Promise();
  process.nextTick( function() {
    p.resolve(f());
  });
  return p;
}
```

Finally, the special • expression is used to allow the execution of callbacks associated with asynchronous computation. Javascript has 'run to completion' semantics, meaning that once a Javascript function is called, that function as well as any other functions invoked synchronously are run without preemption until they terminate. When the special • expression is reached, control returns to the runtime allowing it to flush its work queues according to the semantics described in Section 2.3.

2.2 Priorities

We use priorities to enable the modeling of asynchronous semantics in Node.js using a single abstraction. We are especially interested in the behavior of a regular promise, *.then*, *setTimeout*, *setImmediate*, regular asynchronous I/O, and *process.nextTick*. It turns out we can model all these concepts using a single priority mechanism. We assign the following priorities to the various operations:

0. For *process.nextTick* and regular promises, i.e. the *microtask* queue;
1. For *setImmediate*;
2. For *setTimeout*;
3. For all other asynchronous I/O, e.g. *readFile* etc.

To model all operations as priority promises, we assume an initial heap H_0 that contains the following global promises that are always resolved:

$$\begin{aligned} H_0 = [&\mathit{nexttick} \mapsto (0, \text{res}(\mathit{undef}), []),\\ &\mathit{immediate} \mapsto (1, \text{res}(\mathit{undef}), []),\\ &\mathit{timeout}_0 \mapsto (2, \text{res}(\mathit{undef}), [])] \end{aligned}$$

With these promises we can define various primitives simply as a call to *.then* on one of these predefined promises:

$$\begin{aligned} \mathit{process.nextTick}(f) &= \mathit{nexttick}.\mathit{then}(f)\\ \mathit{setImmediate}(f) &= \mathit{immediate}.\mathit{then}(f)\\ \mathit{setTimeout}(f, 0) &= \mathit{timeout}_0.\mathit{then}(f) \end{aligned}$$

In Section 2.4 we show how general timeouts and I/O is handled.

2.3 Semantics

Figure 4 defines the asynchronous semantics of λ_{async} using priority promises. Reduction rules have the form $H \vdash e \rightarrow H' \vdash e'$ which denotes an expression e under a heap H evaluating to a new expression e' and heap H'. We write $H[p]$ to get the value that an address p points to in the heap H, and $H[p \mapsto v]$ to update the heap H such that p points to value v.

The *evaluation context* E is defined by λ_{js} [11] and basically captures the current execution context. The E-Ctx rule denotes that in any λ_{js} evaluation context E we can evaluate according to λ_{js} (note: in λ_{js} they use σ to denote a heap H), e.g. the premise $H \vdash e \hookrightarrow^* H' \vdash e$ denotes that if we can evaluate and expression e under a heap H to e' with a new heap H' using λ_{js} semantics, we can apply that rule in any evaluation context in our semantics too. All the other rules now deal with just the extension to priority promises.

The E-Create rule creates a fresh promise p in the heap. The priority of a priority promise corresponding to a regular JavaScript promise is always 0, and starts out unresolved with an empty list of callbacks.

The E-Then simply adds a new callback f to the tail of the current list of (as yet unexecuted) callbacks of a promise, where $\oplus$ denotes list append. Note that it doesn't matter whether the promise is currently resolved or not. This behavior corresponds to the promise specification that requires that such function f is never immediately executed even if the promise is already resolved [13, 25.4, 22, 2.2.4].

Rule E-Resolve resolves an unresolved promise by updating its status to resolved (res(v)).

2.3.1 Scheduling

The final rule E-Tick is the most involved and the core of our semantics as it models the scheduler. It takes the 'event loop' expression • and reduces to a sequence of new expressions to evaluate, ending again in •. We discuss each of the three premises in detail:

1. The premise, $R \sqsubseteq \{p \mapsto (n,\ f_1(v); ...; f_m(v)) \mid \forall p \in H, H[p] = (n, \text{res}(v), [f_1, ..., f_m])\}$ selects a set of resolved promises in the set R. The relation $\sqsubseteq$ can be defined in different ways and allows us to discuss different scheduling semantics. For now, we assume $\sqsubseteq$ denotes equality which means we select *all* resolved promises to be scheduled. The resolved promise set R maps each promise to an *expression*, namely a sequence of each callback f_i applied to the resolved value v.

$$\frac{H \vdash e \hookrightarrow^* H' \vdash e}{H \vdash E[e] \;\rightarrow\; H' \vdash E[e']}\ [\text{E-Ctx}]$$

$$\frac{\text{fresh } p}{H \vdash \mathit{Promise}() \;\rightarrow\; H[p \mapsto (0, \text{unres}, [])] \vdash p}\ [\text{E-Create}]$$

$$\frac{H[p] = (n, r, fs)}{H \vdash p.\mathit{then}(f) \;\rightarrow\; H[p \mapsto (n, r, fs \;\oplus\; [f])] \vdash \mathit{undef}}\ [\text{E-Then}]$$

$$\frac{H[p] = (n, \text{unres}, fs)}{H \vdash p.\mathit{resolve}(v) \;\rightarrow\; H[p \mapsto (n, \text{res}(v), fs)] \vdash \mathit{undef}}\ [\text{E-Resolve}]$$

$$\frac{\begin{array}{l} R \;\sqsubseteq\; \{p \mapsto (n,\ f_1(v); \ldots; f_m(v)) \mid \forall p \in H,\ H[p] = (n, \text{res}(v), [f_1, \ldots, f_m])\ \} \\ H' \;=\; H[p \mapsto (n, \text{res}(v), []) \mid \forall p \in R,\ H[p] = (n, \text{res}(v), \_)] \\ R \;\cong\; [p_1 \mapsto (n_1, e_1),\ \ldots, p_m \mapsto (n_m, e_m)] \text{ with } n_k \leqslant n_{k+1} \end{array}}{H \vdash \bullet \;\rightarrow\; H' \vdash e_1; \ldots; e_m; \bullet}\ [\text{E-Tick}]$$

Fig. 4. Priority Semantics of λ_{async}

Note that callbacks are composed *in order* of registration [22,2.2.6.1].

2. Next, the premise $H' = H[p \mapsto (n, \text{res}(v), []) \mid \forall p \in R, H[p] = (n, \text{res}(v), \_)]$ clears all the callbacks for all the promises in the set R so we ensure that a callback never evaluated more than once.
3. Finally, $R \cong [p_1 \mapsto (n_1, e_1), \ldots, p_m \mapsto (n_m, e_m)]$ with $n_k \leqslant n_{k+1}$ denotes that the set R is isomorphic to some list with the same elements but ordered by priority. This denotes the order of the final expressions that are scheduled for evaluation next, namely $e_1; \ldots; e_m; \bullet$. It gives implementation freedom to schedule promises of the same priority in any order.

The scheduling relation $\sqsubseteq$ is a parameter to allow for various different scheduling strategies. Clearly the $\sqsubseteq$ relation must at least be a subset relation $\subseteq$ but by making it more restrictive, we obtain various scheduler variants:

All

By having $\sqsubseteq$ be equality we schedule *all* resolved promises at each event loop tick (in priority order). This is a nice strategy as it has simple declarative semantics and also prevents I/O starvation that we saw in earlier examples: even when we recurse in a *process.nextTick*

Micro

Generally, promises (and *process.nextTick*) are implemented using queues independent of the I/O event manager and programmers can rely on those being evaluated before the callbacks associated with any other I/O operations. We can model this by having a more restrictive $\sqsubseteq$:

1. If there are any resolved promises of priority 0, select just those.
2. Otherwise select all resolved promises as in All.

This strategy already nicely explains the behavior of the two examples given in Section 1.1: the first example uses `process.nextTick` recursively and thus the I/O action never gets executed since there is always a resolved promise of priority 0. However, the second example with a recursive `setImmediate` leads to just promises of priority $\geqslant 1$ and the I/O action can execute too.

Edge

The Edge browser implements promises using the timer queue. This means it is like Micro except that the $\mathit{timeout}_0$ promise has priority 0 too (and *immediate* does not exist).

Node

Node.js is a further restriction of the Micro strategy where micro tasks can run even in between other tasks:

1. If there are any resolved promises of priority 0, select *one* of those.
2. Otherwise select *one* other resolved promise (regardless of its priority).

2.3.2 Node.js Scheduling

The Node scheduling strategy describes strictly *more* possible schedules than are observable in the actual Node.js implementation. In particular, Node.js picks a bit more specific than just any resolved promise in case *2*. After studying the documentation [18], the actual implementation code, and running several tests, we believe Node.js currently schedules using 3 phases:

1. run all resolved immediates (i.e. promises of priority 1);
2. run all resolved timers (i.e. promises of priority 2);

3. run a fixed number of resolved I/O tasks (i.e. promises of priority 3).

and keep running recursively all resolved priority 0 promises in between each phase. However, the *process.nextTick* and *.next* callbacks are scheduled in order. So in between each phase:

a. run all resolved *process.nextTick* promises recursively
b. run all resolved *.next* promises recursively
c. keep doing this until there are no resolved priority 0 promises left

The Node scheduling strategy always includes this more constrained scheduling as implemented by Node.js (but allows more possible schedules). One point where the difference shows up is when there are several resolved promises of different priorities. In that case Node.js will always run them in priority order according to the phases while our Node scheduling allows any order. However, this cannot be reliably observed since the resolving of promises with priority $\geqslant 1$ is always non-deterministic. As such, we believe that Node faithfully models any observable Node.js schedule.

An example of a program that could exhibit an observable difference is:

```
function rec() { setImmediate(rec); }
setImmediate(rec);
setTimeout(f,0);
```

The above program calls `setImmediate` recursively. In the Node.js implementation the timeout will at some point be scheduled and `f` will run. According to our Node strategy this is one possible strategy, but it is also possible to keep recursing on `setImmediate` forever.

A dual situation where the difference is apparent is in the phased scheduling of *process.nextTick* and regular promises. For example:

```
var p = Promise.resolve(42).then( function(v) {
  console.log(v);
});
function rec(){ process.nextTick(rec); }
rec()
```

In Node.js this recurses indefinitely in `rec` and never prints `42` to the console. Under the Node semantics this is a legal strategy but it is also allowed to pick the promise for scheduling and interleave it with the `nextTick`.

We believe that the current situation is not ideal where the actual scheduling strategy of Node.js is too complex and where it is not clear what invariants programmers can expect. In our opinion it would help the community to clarify what invariants can be expected from the schedule. In particular, we feel that the Micro strategy could be a good candidate to consider for helping programmers to reason about scheduling behavior: this is a relatively simple strategy that clearly explains all tricky examples presented in this paper, preserves high-priority scheduling for the micro task queue and promises, and still allows for efficient scheduling implementations.

2.4 Modeling Asynchronous I/O

Up till now we have only modeled deterministic operations like `process.nextTick`, `setImmediate`, `setTimeout(f,0)`, and `.then`, but we did not model arbitrary timeouts or other I/O operations such as `readFile`. We can model this formally with an extra map O of *outstanding* I/O events that maps external I/O events ev to a list of waiting promises. Figure 5 adds three new transition rules of the form $O, H \vdash e \longrightarrow O', H' \vdash e$. We assume that the initial O contains all possible events mapped to an empty list of promises.

The first rule E-Events extends all our previous transition rules to also apply under an event map O of outstanding events and that those rules always leave the event map unchanged.

The next rule E-Register defines register(ev, n), which creates a new promise p of priority n that is resolved whenever the event ev occurs. It extends the mapping of I/O events O with $[ev \mapsto ps \oplus [p]]$ to remember to resolve p when ev happens.

The final rule E-Oracle is an *oracle* and can be applied at will to resolve any event ev with some fresh value v, and resolves all promises waiting for ev to value v. This is the rule that basically models the external world as we can trigger it at any time with any resolved value.

We can now implement various primitives as instances of *register* as:

$$readFile(name, f) = \mathsf{register}(readFile(name), 3).then(f)$$
$$setTimeout(f, ms) = \mathsf{register}(timeout(now() + ms), 2).then(f)$$

where $readFile(name)$ and $timeout(ms)$ are event ev instances.

3 Example Application – Race Detection

To illustrate the importance of accurately modeling the event-loop execution semantics we examine the design of an async-race detector. Although Node programs always run on a single thread and run each callback to completion it is still possible to create data-races between several callbacks [8], [17] that all access the same resource and may have their executions interleaved. To check for races we combine our semantics of async scheduling from Section 2 with a method for exploring possible execution interleavings which could be either static [15] or dynamic such as [16], [8], or [9]. If we find a case with a use/mod or mod/mod conflict we can report it to the user. In such a tool it is critical to have a correct and precise model of the underlying async scheduling semantics to avoid either missing potential issues or reporting many spurious warnings. The following example shows a case where using a simplified model of the Node.js event scheduling will result in a spurious warning:

```
var x = undefined;
setImmediate(function() {
```

$$\frac{H \vdash e \;\longrightarrow\; H' \vdash e'}{O, H \vdash e \;\longrightarrow\; O, H' \vdash e'}\;[\text{E-Events}]$$

$$\frac{\text{fresh } p \quad O[ev] = ps}{O, H \vdash \text{register}(ev, n) \;\longrightarrow\; O[ev \mapsto ps \oplus [p]],\; H[p \mapsto (n, \text{unres}, [])] \vdash p}\;[\text{E-Register}]$$

$$\frac{O[ev] = [p_1, \ldots, p_m] \quad \text{fresh } v \quad \text{some } ev}{O, H \vdash e \;\longrightarrow\; O[ev \mapsto []],\; H \vdash p_1.resolve(v); \ldots; p_m.resolve(v);\; e}\;[\text{E-Oracle}]$$

Fig. 5. Modeling asynchronous I/O events.

```
  console.log(x.f);
});
process.nextTick(function() {
  x = { f: 'hello world' };
});
```

If we were to use a simple model for possible async callback scheduling in Node, say where there is a single task queue, then for the following code we would report that the `nextTick` and `setImmediate` callbacks could happen in either order. Clearly, executing the `setImmediate` callback first will result in a property access to an `undefined` value. However, from our semantics in Section 2 we know that the microtask queue with the `nextTick` callback *must* always be drained before the regular timer events where the `setImmediate` callback is placed. Thus, using the semantics in Section 2 we will correctly conclude there are no data-races in this example, and also, if the `nextTick` is replaced by a `setImmediate` then correctly report that there is a potential race.

4 Asynchronous Execution Context Definition

This section defines generalized concepts around asynchronous execution that abstract many of the low-level queuing and dispatch details in Section 2 and simplify reasoning about the relationships between the executions of asynchronous callbacks.

4.1 Context Definitions

In practice there are multiple distinct concepts of *asynchronous context* that developers use to understand the behavior of an application.

We begin with a basic definition of *invocation indexing* that allows us to distinguish between multiple dynamic callback invocations of the same function definition. Given a function definition `f` we denote the i^{th} dynamic callback invocation of any function during the programs execution with $\texttt{f}^i$. This invocation index notation allows us to distinguish between invocations of the same function definition at different points in a single asynchronous execution chain or in distinct execution chains such as in:

```
let ctr = 0;
function f() { console.log(ctr++ === 0 ? 'hi' : 'bye'); }
setTimeout(f, 10);
setTimeout(f, 20);
```

Using the invocation index notation on this code produces $\texttt{global}^1$ for the first and only execution of the global scope code, produces $\texttt{f}^2$ on the execution of `f` that results from the `setTimeout` with delay of 10 which prints "hi", and finally produces $\texttt{f}^3$ on the second execution of `f` that results from the `setTimeout` with delay of 20 which prints "bye".

Using this indexing we can define two fundamental context relations, *linking context* and *causal context*, on invocations of functions, `f` and `g`, in an asynchronous execution as follows:

- **linking context:** relates $\texttt{f}^i$ *links* $\texttt{g}^j$ if during the execution of $\texttt{f}^i$ the function $\texttt{g}^j$ is linked to a priority promise via an E-Then rule application.
- **causal context:** relates $\texttt{f}^i$ *causes* $\texttt{g}^j$ if during the execution of $\texttt{f}^i$ the function $\texttt{g}^j$ is enabled for execution via an E-Then rule application on an already resolved priority promise or via an E-Resolve rule application on a previously un-resolved priority promise.

For programs that do not use JavaScript promises and rely on raw callbacks the linking context and causal context will be the same at all program points. However, promises can be linked and resolved at different points in the code and thus the context relations may differ:

```
function f(val) { console.log(val); }
let p = new Promise();
p.then(f);

function resolveIt() { p.resolve('hi'); }
setImmediate(resolveIt);
```

In this case we will have $\texttt{global}^1$ *links* $\texttt{f}^3$ due to the `.then` in the global scope execution but $\texttt{resolveIt}^2$ *causes* $\texttt{f}^3$ since it does the actual resolution that results in the `f` being enabled for execution and, eventually, executed.

Finally, we note that due to the definition of *invocation indexing* we have a total order for the temporal *happens before* relation on functions where $\texttt{f}^i$ *happens before* $\texttt{g}^j$ *iff* $i < j$.

4.2 Lifting Context Relations to Chains

In the previous section we defined the binary relations for linking context and causal context in asynchronous execution flows. We note that the linking context and causal context relations are both *transitive*. That is, if $\texttt{f}^i$ *causes* $\texttt{g}^j$ and $\texttt{g}^j$ *causes* $\texttt{h}^k$ then we can infer $\texttt{f}^i$ *causes* $\texttt{h}^k$ as well with the same property holding for linking context and *links*. Similarly, we note that both relations form a tree structure over the indexed invocations in the program. As a result, applications that require global information about an asynchronous execution chain, or multiple links in a relation, can walk the relation tree to extract the desired information.

Analyzing the transitive closure of these relations provides important diagnostic information such as counting the number of CPU cycles used to service a single http request or computing *long call stacks* [25]. Starting from the top-level request handler of a web application, we can recursively aggregate all of the CPU time of functions that are transitively related to the handler by the causal context relation to compute the total time used to handle the request. To compute the long call-stack at a particular program point, we can traverse the inverse linking context relation (traverse up the tree) stitching together the call-stacks at each point to produce the long call-stack for that point in the execution. In our example program above, if we want a long call-stack starting at the `console.log(val)` statement, we would first collect the short call stack at that point which includes `console.log` and `f`. Then, because $\texttt{global}^1$ *links* $\texttt{f}^3$ in the linking context relation so we would follow the inverse up the tree to find the stack associated with the $\texttt{global}^1$ invocation. This short call stack would include `global` as well as `p.then`. These two call-stacks can then be stitched together to produce the desired long call-stack.

4.3 Computing Context Relations

To compute the linking context and causal context relations we begin by introducing the following helper functions to handle the management of relational information during the processing of async callbacks.

```
let currIdxCtx = undefined;
let linkRel = new Map();
let causalRel = new Map();

bindLink(e) {
  return {exp: e, linkCtx: currIdxCtx};
}

bindCausal(linke) {
  return Object.assign({causalCtx: currIdxCtx}, linke);
}

unpack(bounde) {
  currIdxCtx = genNextCtxIdx(bounde.exp);

  let lr = linkRel.get(bounde.linkCtx) || [];
  lr.push(currIdxCtx);
  linkRel.set(bounde.linkCtx, lr);

  let cr = causalRel.get(bounde.causalCtx) || [];
  cr.push(currIdxCtx);
  causalRel.set(bounde.causalCtx:, cr);

  return bounde.exp;
}
```

These functions use a global `currIdxCtx` to track the invocation index of the currently executing code. The `bindLink` function takes an expression being passed `.then` and creates a new record with the expression value being linked and sets the linking context to the invoke index of the currently executing code. Similarly, `bindCausal` which can only happen after the linking has occurred, takes the record and updates it with the index of the currently executing code as the `causalCtx` value. Finally, the `unpack` function sets a fresh invocation index for the code that is about to be executed and unpacks the link and causal invoke index values to update the corresponding linking context and causal context relations for the callback.

These helpers can be integrated into the E-Then and E-Resolve rules from Figure 4 to produce the rules shown in Figure 6. We begin by altering the list of pending callbacks, *fs*, from Figure 4 to instead be a list of bound records. We also explicitly split out the cases of a `.then` into E-Then-Unresolved which triggers when a callback is linked with an *unresolved* promise via `then` and E-Then-Resolved which triggers when a callback is linked to a promise that has already resolved. In the first case the priority promise is unresolved so we only set the context record with the invoke index for the link time using the `bindLink` helper function. In the case where the priority promise is already resolved we update the link and the causal context since the callback is immediately eligible for execution. The E-Resolve-Ctx rule is updated to fill in the empty causal record field, using the current invoke index, using the `bindLink` function.

The E-Tick rule is updated to set the asynchronous context to the appropriate value and to update the linking context and causal context relations. To accomplish the updates the E-Tick-Ctx rule sets each expression to be evaluated as $\mathit{unpack}(be_i)(v)$, which when evaluated, will perform the update to the current execution index, update the relations as needed, and the invoke the desired callback function with the resolved priority promise value.

4.4 Example

```
(function foo() {
  const p = new Promise(function promise1(res) {
    setTimeout(function timeout1() {
      res(42);
    }, 200);
  });

  setImmediate(function immediate1() {
    p.then(function then1(val) {
```

$$\frac{H[p] = (n, \text{unres}, fs)}{H \vdash p.then(f) \;\rightarrow\; H[p \mapsto (n, \text{unres}, fs \;\oplus\; [bindLink(f)])] \vdash undef} \;[\text{E-Then-Unresolved}]$$

$$\frac{H[p] = (n, \text{res}(v), fs)}{H \vdash p.then(f) \;\rightarrow\; H[p \mapsto (n, \text{res}(v), fs \;\oplus\; [bindCausal(bindLink(f))])] \vdash undef} \;[\text{E-Then-Resolved}]$$

$$\frac{H[p] = (n, \text{unres}, fs)}{H \vdash p.resolve(v) \;\rightarrow\; H[p \mapsto (n, \text{res}(v), [bindCausal(linke) \mid linke \in fs])] \vdash undef} \;[\text{E-Resolve-Ctx}]$$

$$\frac{\begin{array}{l} R \;\sqsubseteq\; \{p \mapsto (n,\; unpack(be_1)(v); \ldots; unpack(be_m)(v)) \mid \forall p \in H,\; H[p] = (n, \text{res}(v), [be_1, \ldots, be_m])\} \\ H' \;=\; H[p \mapsto (n, \text{res}(v), [])] \mid \forall p \in R,\; H[p] = (n, \text{res}(v), \_)] \\ R \;\cong\; [p_1 \mapsto (n_1, e_1),\; \ldots, p_m \mapsto (n_m, e_m)] \text{ with } n_k \leqslant n_{k+1} \end{array}}{H \vdash \bullet \;\rightarrow\; H' \vdash e_1; \ldots; e_m; \bullet} \;[\text{E-Tick}]$$

Fig. 6. Introduction of Linking and Causal Context

```
      console.log('Hello Context World!');
    });
  });
})();
```

First, we observe that as a result of the event loop priorities described earlier in this paper, this code will result in the invocation indexed sequence (where → indicates direct invocation and ⇒ indicates a fresh turn of the event loop): global^1 → foo → Promise → promise1 → setTimeout → setImmediate ⇒ immediate1^2 → Promise.then ⇒ timeout1^3 → res ⇒ then1^4 → console.log

We assume that the first turn of the event loop has context `global`^{1}. From program start, execution will proceed synchronously until `setTimeout` is reached. At this point, `timeout1` is both causal and link bound to produce the record `{exp: timeout1, linkCtx: global`^{1}`, causalCtx: global`^{1}`}`. Execution then continues to `setImmediate` where `immediate1` will be similarly causally and link bound produce the record `{exp: immediate1, linkCtx: global`^{1}`, causalCtx: global`^{1}`}`. On the next turn of the event loop, we unpack and execute the bound `immediate1` function setting the current context to `immediate1`^{2} and adding `global`^{1} *links* `immediate1`^{2} and `global`^{1} *causes* `immediate1`^{2} to the relations. Next we reach the call to `p.then` which performs only the link binding, since `p` is unresolved, to produce the record `{exp: then1, linkCtx: immediate1`^{2}`}`. Once the timer fires, `timeout1` will be unpacked setting the current context to `timeout1`^{3} and adding `global`^{1} *links* `timeout1`^{3} and `global`^{1} *causes* `timeout1`^{3} to the relations. Promise `p` then resolves updating the record for `then1` with causal context `timeout1`^{3}. Finally, `then1` is unpacked introducing context `then1`^{4} and adding `immediate1`^{2} *links* `then1`^{4} and `timeout1`^{3} *causes* `then1`^{4} to the relations.
The linking context and causal context relation trees for the examples are shown in Figure 7. The figure shows that

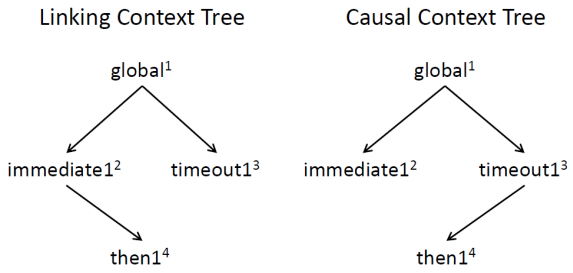

Fig. 7. Linking and Causal Context trees produced for JavaScript example code.

for code callback based API's the linking and causal context relations are the same, in contrast to the promise based operation for `then1`, which has `immediate1`^{2} for the linking context and `timeout1`^{3} for the causal context. If we start from the `console.log` statement in the code, in `then1`^{4}, and build the linking context chain by traversing the parent links we get `then1`^{4} → `immediate1`^{2} → `global1`^{1} while building the causal context chain by traversing the parent links we get `then1`^{4} → `timeout1`^{3} → `global1`^{1}.

4.5 Userspace Queuing

One challenge to providing a comprehensive asynchronous context tracking system in Node.js is the presence of userspace queuing of callbacks. A library writer may want to queue up requests from many different contexts to batch the work for processing. An example of this is a database module that provides access to a remote database and wants to serve all requests through a single connection. In this case all of the callbacks associated with any database request would be stored in the same worklist and then invoked from the shared database context:

```
var pending = [];
```

```
function query(q, cb) {
 pending.push({query: q, cb: cb});
 if (pending.length === BUFFER_SIZE) {
   dbConn.sendQueries(pending);
   pending = [];
 }
}

function process(results) {
  for (var i = 0; i < results.length; i++) {
    var res = results[i];
    res.cb(res.data);
  }
}

var dbConn = openDB('database');
db.on('response', process);
```

In this case, the database wrapper holds callbacks in a userspace queue until enough queries have been buffered at which point it sends them to the database. When the database sends back a response, the wrapper directly invokes all of the callbacks with their associated database results. Unfortunately, this common pattern will result in every `cb` invocation being associated with the same context (the context of the `process` callback).

Fortunately, our design of the helper functions for context tracking in Section 4.3 provides a simple API for developers to use when implementing custom userspace asynchronous queuing. In our example this we would replace the line with `pending.push( {query: q, cb: cb} )` with calls to bind the causal and link contexts `pending.push( {query: q, cb: bindCausal(bindLink(cb))} )`. When this bound callback in invoked later, `res.cb(res.data)`, it will need to be invoked with the unpack helper, `unpack(res.cb)(res.data)`, to set the appropriate information into the current context.

5 Example Application – Time-Travel Debugging

To illustrate how a tool writer can utilize causal context we look at the application of navigating asynchronous context in a time-travel debugging system for Node.js. Operations like reverse-continue are very useful but they move back linearly in time instead of back through the logical execution of calls associated with an asynchronous execution chain which is often what a developer is trying to understand. Consider the example:

```
function tryReadFile(file) {
  fs.readFile(file, function read1(err, data) {
    if(err) /*reverse to here*/
      setTimeout(function timeout1() {
          console.error('error'); /*break*/
      }, 100);
    else
      console.log(data);
  });
}
tryReadFile('foo.txt');
tryReadFile('bar.txt');
```

If the developer is at a breakpoint on the `console.log('error')` line labeled with `/*break*/`then a natural question to ask is "what was the value of `file` that lead to the error". However, since this variable is not closure captured, it is not available when the `setTimeout` callback is executing. If the developer sets a breakpoint at the enclosing *if-statement* (labeled `/*reverse to here*/`) and executes a reverse-continue then they may hit that breakpoint on either the async chain resulting from the read of `foo.txt` or `bar.txt`. Thus, in [4], a *reverse-callback-origin* operation, based on custom async tracking code, is introduced that reverse executes to "the point in time where the currently executing callback was registered".

If the async call chain for `'foo.txt'` when we hit the breakpoint is, `timeout1`^{4} ⟶ `read1`^{2} ⟶ `global1`^{1}, then we know we must reverse-execute to the time when we hit the breakpoint at `/*reverse to here*/` *and* the currently executing callback `currIdxCtx` is `read1`^{2}. If we hit the breakpoint in another context then we know we are encountering this breakpoint on a different async call chain, in our example corresponding to the read from `'bar.txt'`, and should ignore the breakpoint.

6 Example Application – Resource & Priority Scheduling

As Node applications become more sophisticated there is an increasing need for more powerful scheduling mechanisms than the assortment of special purpose queues currently provided. It is common for Node.js web-servers to intermittently download updated content from a master source and upload telemetry data to a centralized server. These actions are low-priority and do not have hard deadlines. However, servicing a user request for content is both high-priority and must be low latency to ensure a good experience. Currently, there is no mechanism to specify that a task is low priority or that it may involve heavy I/O activity. Thus, programs risk blocking high priority activity, seriously impacting service quality. While some scheduling behavior can be controlled using carefully written yields to manually spread out background processing workloads, others cannot be expressed at all today in Node.js.

To address these scenarios and simplify the semantics of the multi-queue implementation in Section 2 we propose a *resource & priority aware soft-realtime* implementation for the Node.js event loop using the following design goals:

1. Priority based scheduling using a single priority-ordered event queue and an extended range of *priorities*.
2. Support for task *deadlines* to enable soft realtime scheduling, and promotion of priorities to prevent starvation of background tasks.
3. Support for manual priority specification as well as automatic priority inheritance. We extend core Node

API's to take optional explicit priorities and support automatic propagation of priorities via *async-context*.
4. Resource awareness and scheduling using expected I/O, including network and disk, plus time information.
5. Online learning of resource use for async-calls. We dynamically build an estimate of the CPU and I/O resources used by each individual callback and, using *async-context*, the total cost of each request (http requests or other labeled request trigger).

Priority and Deadline. Our first task is to extend the representation of *priority promises* from Section 2 with an basic set of *soft-priority* levels:

```
'high' > 'low' > 'background'
```

These new priority levels allow us to add additional constraints to the scheduling relation such that we will schedule higher priority tasks before lower levels.

Next we extend the representation with information on an optional target completion *deadline* by which the task should be completed. This allows the scheduler to perform soft-realtime scheduling. Further, we can ensure that a low or background task is not starved as we can look at the `deadline` and promote the priority if needed.

This design allows us to define existing API's and new API's in a uniform manner for the scheduler as:

```
runTask(cb, level) //priority=level, deadline=0
runSchedule(cb, time) //priority=high, deadline=time
runBackground(cb, time) //priority=background, deadline=time
```

The new classes of tasks include running multiple tasks of different priority via `runTask`, running a task with a soft completion deadline via `runSchedule`, and running a background task that will increase in priority once the given deadline expires via `runBackground`.

One challenge is that in order to allow mixing of new context aware and old context unaware API's we need to support automatic propagation of priority levels across async-callbacks. In cases of simple callbacks, such as `setImmediate` or `setTask` without providing a priority level there is a single choice of taking the priority of the currently executing callback since the *causal context* and *linking context* contexts are the same. However, in the case of promises these parent contexts are different and we could could choose to inherit priority from either. Consider the example:

```
let p = new Promise((resolve, reject) => {
  //Low priority promiseified file read
  fs.readFile('foo.txt', function(err, data) {
    resolve(data);
  }, 'low');
});

//Inherit from link parent (high) or from causal parent (low)?
p.then((data) => { console.log(data); });
```

Using causal context would result in the `then` function being run in low priority regardless of how the priority is setup when the `.then` is registered. Thus, our design uses priority inheritance based on linking context to avoid unintentionally lowering the priority of a callback.

Resource Use and Automatic Learning. Scheduling requires an estimate of the time and I/O resources needed by a task. Instead of requiring a developer to explicitly provide this we can automatically gather this by dynamically monitoring execution behavior and aggregating resource usage. Our proposed design uses a distribution, mean and standard deviation, for the expected time, network, and disk loads of a callback. We compute both the inclusive and exclusive values so the scheduler can use both the immediate and overall cost of a task to make a decision.

```
type dist = { mean: number, stdev: number };

type resourceEstimate = {
  time: { inclusive: dist, exclusive: dist },
  network: { inclusive: dist, exclusive: dist },
  disk: { inclusive: dist, exclusive: dist }
}
```

To compute these resource requirements we use causal context semantics from Section 4 to aggregate the various costs to the *parent* callbacks. In this model each callback tracks time & I/O when it is directly executing to compute the exclusive distribution. At then end of an async execution we can use the async causal context tree to aggregate for the inclusive resource use distribution.

This design proposal demonstrates how the semantics defined in this paper (1) can be used in the design of a new async execution implementation that preserves fundamental Node semantics while enabling new functionality and (2) how our definitions of *context* can be used in this design process to inform the behavior and design of the new implementation.

7 Related work

The Node.js runtime is a relatively new platform and, as a result, there is limited prior work on the semantics of the Node.js execution model and diagnostic tools for it. For the most part academia has focused on the idea of race-detection in asynchronous code while industry has tended to focus on the topic of application performance management (APM) tooling.

JavaScript and Node.js Semantics. [11] presents a core calculus for the procedural JavaScript language λ_{js} which is extended by [15] to include a partial formalization of the asynchronous semantics for Node.js but omits the recently added ES6 Promise semantics and uses a simplified model for the drain rules for the event queues in Node.js. Work in [3], uses an implicitly defined model for asynchronous execution that is encoded in the dynamic analysis and rules for ''Callback scheduling'' and ''Callback invocation''. Other work has utilized concrete implementation details of a specific version of LibUV and Node.js, [8] uses the deprecated

v0.12.7 version of Node.js, and the details of the event-loop semantics are implicitly encoded in the tool implementation.

Asynchronous Semantics. There is much work on cooperative threads going from concurrency primitives in ML [5] to mixing cooperative threads with preemptive ones [7]. A recent semantics for cooperative threads was described by Boudol [6] and later by Abadi and Plotkin [1] in terms of algebraic effects [20, 21]. Leijen [14] describes an implementation of `async/await` in terms of algebraic effects on top of Node.js asynchrony.

Race Detection. Data races in Node.js are of particular interest to the academic community and have been approached from both a static [15] and dynamic analysis viewpoint [17], [8]. The work in [15] presents a number of case studies for identifying race conditions in real-world bug reports but, as shown in Section 3 the simplified model of asynchronous execution semantics can lead to false positives. From the dynamic viewpoint [8] uses a model of the event loop implementation in Node.js v0.12.7 to fuzz possible execution schedules to expose race related bugs. However, as explored in [17], certain data races may be benign and the ability to focus on what are likely to be *important* issues is critical, e.g., by focusing on fuzzing certain categories of non-determinism as discussed in Section 4 of [8], and is a topic that requires more investigation. Going beyond simply detection the work in [2] looks at forcing an application to use a known good schedule to prevent race related failures from manifesting themselves.

Application Performance Monitoring Tooling. Industrial users of Node.js place a high value of the availability and performance of their services. As a result there is a large investment in application performance monitoring (APM) tooling including [10], [19], [24], and [23]. These systems will dynamically instrument running applications to track errors, upload logging output, and extract information on request handling time statistics among other performance and stability data. A key component in these systems is correlating data with the request (usually http) that was responsible generating it. For example if a request generates an error or takes an abnormally long time then they want to show all the log statements or time-stamps from the *async-contexts* associated with that request. Thus, tracking async-context is critical for all of these systems but, in the absence of a formal definition, all of these systems implement custom ad-hoc tracking based on the particular needs of their tooling.

Asynchronous Context Tracking Although asynchronous context is a fundamental concept in Node.js programming it has not previously been explicitly formalized. Prior research work such as [3], tooling such as [10] or [25], and runtime enhancement proposals such as [12] or [26] have all informally defined and used a notion of asynchronous context. In many cases this definition was closely tied to the specifics of the problem they were addressing, [25], or as in the case of the initial [12] proposal, closely tied to the specifics of a particular implementation feature.

The concept of an *Event-Based Call Graph* is introduced in [15] to track relations between callback definitions and their invocation, listener registration, or scheduling. However, in contrast to the linking context/causal context definitions for asynchronous context in this work, event-based call graphs do not differentiate between different dynamic execution chains over the same set of asynchronous function declarations. The example in Section 5 illustrates this via the two different calls with different argument data to the `readFile` function which, from a developer perspective, represent two logically distinct asynchronous execution chains. The definitions in [15] Section 4.1 create a single node corresponding to the callback declaration passed to `readFile` with a single *call edge* for both the `foo.txt` and `bar.txt` invocations which result in the asynchronous execution chains through the `setTimeout` callback being merged as well. Conversely, as described in Section 5, the distinct asynchronous execution chains for these two calls are preserved by the linking context/causal context relations defined in this work.

8 Conclusion

This paper presented a formalization of the Node.js event-loop and asynchronous execution semantics in Section 2 as well as the definition of high-level concepts around asynchronous execution context in Section 4. Beyond the utility of providing a well defined model for other researchers or practitioners to use, the formalization of these definitions provides insight into key features of the Node.js execution model. Our formulation of the Node.js event-loop demonstrates that by adopting a novel *priority promise* based view we can cast the multiple event-queues in the current Node.js implementation as a single work-list with a parameterizable set of scheduling constraints. This view allows us to easily express and compare several schedule models and leads naturally to the more sophisticated fully priority and resource aware scheduling design we explored in Section 6. This work also defines the distinct but related types of context – causal context and linking context – that are fundamental to understanding asynchronous program execution. Thus, the formalizations presented in this paper provide a foundation for research into static and dynamic program analysis tools, support the exploration of alternative Node.js event loop implementations, and provide a high-level conceptual framework for reasoning about relationships between the execution of asynchronous callbacks in a Node.js application.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and constructive feedback. We also appreciate the discussions and feedback from our colleagues Bhaskar Janakiraman, Mike Kaufman, Ali Sheikh, and Jimmy Thomson.

References

[1] Martín Abadi, and Gordon D. Plotkin. "A Model of Cooperative Threads" 6 (4:2): 1–39. 2010. doi:10.2168/LMCS-6(4:2)2010.

[2] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. "Repairing Event Race Errors by Controlling Nondeterminism." In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 2017. doi:10.1109/ICSE.2017.34.

[3] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Understanding Asynchronous Interactions in Full-Stack JavaScript." In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 2016. doi:10.1145/2884781.2884864.

[4] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. "Time-Travel Debugging for JavaScript/Node.js." In *Proceedings of the 2016 ACM International Symposium on the Foundations of Software Engineering (FSE'16)*. Nov. 2016. doi:10.1145/2950290.2983933.

[5] Dave Berry, Robin Milner, and David N. Turner. "A Semantics for ML Concurrency Primitives." In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 119–129. POPL'92. Albuquerque, New Mexico, USA. 1992. doi:10.1145/143165.143191.

[6] Gérard Boudol. "Fair Cooperative Multithreading." In *Concurrency Theory: 18th International Conference*, edited by Luís Caires and Vasco T. Vasconcelos, 272–286. CONCUR'07. Lisbon, Portugal. Sep. 2007. doi:10.1007/978-3-540-74407-8_19.

[7] Frédéric Boussinot. "FairThreads: Mixing Cooperative and Preemptive Threads in C." *Concurrent Computation: Practice and Experience* 18 (5): 445–469. Apr. 2006. doi:10.1002/cpe.v18:5.

[8] James Davis, Arun Thekumparampil, and Dongyoon Lee. "Node.Fz: Fuzzing the Server-Side Event-Driven Architecture." In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. 2017. doi:10.1145/3064176.3064188.

[9] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. "Systematic Testing of Asynchronous Reactive Systems." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE'15)*. 2015. doi:10.1145/2786805.2786861.

[10] Glimpse. 2016. `http://node.getglimpse.com`.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of Javascript." In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*. 2010. doi:10.1007/978-3-642-14107-2_7.

[12] Async Hooks. 2016. `https://github.com/nodejs/diagnostics/tree/master/tracing/AsyncWrap`.

[13] Ecma International. 2015. `https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects`.

[14] Daan Leijen. *Structured Asynchrony Using Algebraic Effects*. MSR-TR-2017-21. Microsoft Research. May 2017.

[15] Magnus Madsen, Frank Tip, and Ondřej Lhoták. "Static Analysis of Event-Driven Node.js JavaScript Applications." In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*, 505–519. Oct. 2015. doi:10.1145/2858965.2814272.

[16] Madanlal Musuvathi, and Shaz Qadeer. "Fair Stateless Model Checking." In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. 2008. doi:10.1145/1375581.1375625.

[17] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. "Detecting JavaScript Races That Matter." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE'15)*. 2015. doi:10.1145/2786805.2786820.

[18] Node.js Foundation. 2017. `https://nodejs.org/api/process.html#process_process_nexttick_callback_args`.

[19] NSolid. 2015. `https://nodesource.com/products/nsolid`.

[20] Gordon D. Plotkin, and John Power. "Algebraic Operations and Generic Effects." *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

[21] Gordon D. Plotkin, and Matija Pretnar. "Handlers of Algebraic Effects." In *18th European Symposium on Programming Languages and Systems*, 80–94. ESOP'09. York, UK. Mar. 2009. doi:10.1007/978-3-642-00590-9_7.

[22] Promise/A. "Promise/A+ Specification." 2014. `https://promisesaplus.com`.

[23] New Relic. 2012. `https://newrelic.com/nodejs`.

[24] Stackdriver. 2015. `https://cloud.google.com/trace/`.

[25] Async Call Stacks. 2014. `http://www.html5rocks.com/en/tutorials/developertools/asyn`

[26] Zones. 2014. `https://github.com/angular/zone.js`.