Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Studienprojekt II

Vergleich der parallelen Verarbeitung in den Laufzeitumgebungen Node.js, Deno & Bun

Fabian Friedrich

vorgelegt am 20. Februar 2024

•

Fachbereich Duales Studium Wirtschaft / Technik
Hochschule für Wirtschaft und Recht Berlin

Name:	Fabian Friedrich
Ausbildungsbetrieb:	Bosch Siemens Hausgeräte GmbH
Fachrichtung:	Informatik
Studienjahrgang:	2021
Betreuer Hochschule:	Arthur Zimmermann
Betreuer Unternehmen:	Nicolina Warnhoff
Wörter:	4152

Abstract

This study paper focuses on comparing parallel processing in JavaScript using the runtime environments Node.js, Deno, and Bun. The paper includes a detailed description of the development of JavaScript, JavaScript engines and the mentioned runtime environments. A major emphasis is placed on analyzing asynchronicity in JavaScript through the use of callbacks and promises, as well as the implementation of worker threads in the runtime environments. Through benchmarking, the runtime environments are compared based on criteria such as runtime, CPU usage, and RAM consumption. The results indicate that Node.js exhibits the highest resource demands and longest execution times, whereas Bun emerges as the fastest runtime environment with lower resource consumption. An ultimate evaluation of Deno was not possible due to a bug in the thread management of Deno.

Zusammenfassung

Diese Studienarbeit widmet sich einem Vergleich der parallelen Verarbeitung in JavaScript unter Verwendung der Laufzeitumgebungen Node.js, Deno und Bun. Die Arbeit umfasst eine ausführliche Beschreibung der Entwicklung von JavaScript sowie JavaScript-Engines und der genannten Laufzeitumgebungen. Ein Schwerpunkt liegt auf der Analyse der Asynchronität in JavaScript durch die Verwendung von Callbacks und Promises sowie der Implementierung von Worker-Threads in den Laufzeitumgebungen. Im Rahmen eines Benchmarks werden die Laufzeitumgebungen anhand von Kriterien wie Laufzeit, CPU-Auslastung und RAM-Verbrauch miteinander verglichen. Die Ergebnisse zeigen, dass Node.js die höchsten Ressourcenanforderungen aufweist und die längsten Ausführungszeiten erzielt, während Bun als schnellste Laufzeitumgebung hervorgeht und dabei weniger Ressourcen verbraucht. Eine abschließende Bewertung von Deno war aufgrund eines Fehlers im Thread-Management nicht möglich.

Inhaltsverzeichnis

1 Einleitung

JavaScript, eine Skriptingsprache die speziell für das Web entwickelt wurde, ist seit 11 Jahren in Folge die meistgenutzte Programmiersprache. [1] Das Web verzeichnete in den vergangenen Jahren ebenfalls ein signifikantes Wachstum und es ist wahrscheinlich, dass dieser Trend in den kommenden Jahren weiterhin anhalten wird.
Aufgrund der hohen Popularität der Sprache [1] werden viele neue Tools und Technologien rund um die Sprache entwickelt. Darunter Laufzeitumgebungen, die ermöglichen, JavaScript auch außerhalb des Webs zu verwenden. Zu diesen Laufzeitumgebungen gehören unter anderem Node.js, Deno und Bun.
Außerdem nimmt die Komplexität von Anwendungen stetig zu, was ebenfalls eine steigende Ressourcennachfrage mit sich bringt. Gleichzeitig werden häufig mehrere Anwendungen parallel genutzt, wobei diese Anwendungen selbst vermehrt Aufgaben parallel verarbeiten, um eine schnellere Laufzeit zu gewährleisten.

Diese Studienarbeit fokussiert sich darauf, die Hintergründe von JavaScript sowie Laufzeitumgebungen, insbesondere Node.js, Deno und Bun, zu erläutern. Außerdem werden die Aspekte der parallelen Verarbeitung in JavaScript und den genannten Laufzeitumgebungen detailliert erläutert, durch Benchmarks bewertet und miteinander verglichen.

2 Grundlagen

Das kommende Kapitel bietet einen umfassenden Einblick über die Entwicklung von JavaScript, JavaScript-Engines und Laufzeitumgebungen wie Node.js, Bun und Deno. Außerdem werden die Konzepte von Prozessen, Threads, paralleler Verarbeitung und der Asynchronität in JavaScript und den genannten Umgebungen erläutert.

2.1 Die Entwicklung und Bedeutung von JavaScript

JavaScript ist eine Skriptingsprache, welche für das Web von Brendan Eich im Jahre 1995 bei NetScape entwickelt wurde. JavaScript wurde initial für den Webbrowser „NetScape Navigator 2.0" entwickelt. [2] [3] Zum Zeitpunkt der Erstveröffentlichung hieß JavaScript noch LiveScript. Allerdings wurde der Name innerhalb von drei Monaten bekanntermaßen von LiveScript zu JavaScript geändert. [2] Der Name wurde gewählt, da Java zu dem Zeitpunkt eine sehr beliebte Programmiersprache und somit eine Inspiration für JavaScript war. Dabei wurde der Versuch unternommen, die positiven Verbindungen, die mit Java in Verbindung gebracht wurden, auch auf JavaScript zu übertragen, um von dessen Popularität zu profitieren. Bis auf den Namen hat Java allerdings nicht sehr viel mit JavaScript zu tun. [2] [4]

JavaScript wird heute von allen modernen Browsern unterstützt und verwendet. Durch JavaScript ist es möglich geworden, Webseiten interaktiv zu gestalten und ganze Programme für Webbrowser zu bauen. [4] Damit alle Browser, die angeben JavaScript zu unterstützen, auch wirklich echtes JavaScript meinen, gibt es den ECMAScript-Standard. [5] Dieser Standard gibt Regeln für die Skriptingsprache ECMAScript vor. Webbrowser halten sich bei der Unterstützung von JavaScript an diesen ECMAScript-Standard. ECMAScript und JavaScript sind somit nahezu identische Sprachen und können synonym verwendet werden. [4] [3]

2.2 Die Funktionsweise und Entwicklung von JavaScript-Engines

Eine JavaScript-Engine ist ein Programm, welches JavaScript Code versteht und in von Maschinen lesbaren Code umwandelt. Jedes Programm, das JavaScript benutzen möchte, braucht eine JavaScript-Engine. [6] Somit implementieren auch Webbrowser ihre eigenen JavaScript-Engines. Im Chromebrowser von Google wird zum Beispiel die V8-Engine eingesetzt. Bei Firefox wird SpiderMonkey verwendet und der Safaribrowser benutzt die JavaScriptCore-Engine. [6]

JavaScript ist eine interpretierte Sprache. Das bedeutet, der Programmcode muss nicht vor der Verwendung des Programms in Maschinencode kompiliert werden. Stattdessen werden die einzelnen Programmzeilen zur Laufzeit interpretiert und erst dann in Maschinencode umgewandelt. Interpretierte Sprachen sind dadurch häufig benutzerfreundlicher und ermöglichen eine schnellere Entwicklung. Allerdings sind interpretierte Sprachen dadurch langsamer in der Ausführung. Um die Leistung von JavaScript zur Laufzeit zu verbessern wird ein Just-In-Time, abgekürzt JIT-Compiler, eingesetzt. [7] [6]

Die erste JavaScript-Engine der Welt heißt Mocha und wurde sowie JavaScript selbst von Brendan Eich geschrieben. Diese JavaScript-Engine wurde ebenfalls mit dem „Netscape Navigator 2.0" eingeführt. Mocha war ein einfach gehaltener JavaScript-Interpreter. Ein JIT-Compiler wie in heutigen modernen Browsern wurde noch nicht eingesetzt. Dadurch war die Engine auch deutlich langsamer als moderne Engines. Allerdings legte Mocha den Grundstein für die JavaScript-Engine SpiderMonkey in Firefox, die nach wie vor von Mozilla entwickelt und in Firefox verwendet wird. [2]

Just-In-Time-Compilation

JavaScript, sowie andere interpretierte Sprachen sind aufgrund der dynamischen Typisierung und fehlenden Kompilierung in Maschinencode langsamer als stark typisierte und kompilierte Sprachen. Der Just-In-Time-Compiler möchte beide Probleme lösen und verspricht eine Mischung zwischen kompilierten und interpretierten Sprachen. [6]

JIT-Compiler unterscheiden sich in den einzelnen JavaScript-Engines in der Implementierung. Im Grunde genommen funktionieren die meisten allerdings nach dem selben Schema. Zum Start des Programms wird der Programmcode interpretiert.

Das hat den Vorteil, dass Nutzer nicht erst auf die Kompilierung des Programms warten müssen. Stattdessen ist das Programm direkt einsatzbereit. Parallel zur Laufzeit des Programms wird der Code im Hintergrund optimiert. Dabei beobachtet der Compiler den Interpreter. Besonders rechenintensive und häufig aufgerufene Codeteile werden in Maschinencode kompiliert. Somit müssen diese Codezeilen nicht mehr einzeln interpretiert werden, sondern es kann direkt auf den kompilierten Maschinencode zurückgegriffen werden. Der Compiler kompiliert also nicht den gesamten Code zum Start, sondern kompiliert nur die wichtigsten Codestellen bei Bedarf, beziehungsweise „Just-In-Time". [6]

V8-Engine

Google hatte in der Vergangenheit ein Problem mit der Geschwindigkeit von damaligen JavaScript-Engines. Die Vision von Google war es, dass Apps im Webbrowser möglich sind, die sich wie native Programme anfühlen. [8] Deswegen wollte Google 2006 eine neue Engine entwickeln. Dabei sollte besonderen Wert auf die „Geschwindigkeit, Einfachheit, Sicherheit und Stabilität" (aus dem Englischen übersetzt) [8] gelegt werden.

Google stellte Lars Bak ein, um die neue JavaScript-Engine für den Chrome-Browser zu entwickeln. Beim Namen des neuen „JavaScript-Motors" ließ sich das Team um Bak vom sehr leistungsstarken V8-Motor inspirieren, der sonst eigentlich nur in Autos zu finden ist. [9] Die V8-Engine ist nach zwei Jahren Entwicklungszeit im September 2008 veröffentlicht worden. Mit dem Release von V8 wurde der Quellcode veröffentlicht und die Engine wird seitdem von Google und anderen Entwicklern als Open-Source-Projekt weiterentwickelt. [9]

Die V8-Engine ist eine der schnellsten JavaScript-Engines. Die gute Performance der V8-Engine hat verschiedene Gründe. Zum einen ist die Engine nicht in JavaScript sondern in C++ geschrieben. [10] Außerdem implementiert die V8-Engine einen optimierten Interpreter und JIT-Compiler. Wieder inspiriert vom V8-Automotor wird der Interpreter von Google „Ignition" und der JIT-Compiler „TurboFan" genannt. Die V8-Engine führt zusätzliche Optimierungen durch, in dem beispielsweise bestimmte Typenprüfungen vermieden werden. [10]

Die JavaScript V8-Engine von Google wurde initial für den Chrome-Browser entwickelt. Das Entwicklerteam von Google hatte jedoch die Option offen gehalten, dass die V8-Engine auch außerhalb von Browsern eingesetzt werden kann. Im Jahr 2009

wurde die V8-Engine dann in der JavaScript-Laufzeitumgebung Node.js implementiert. [9]

2.3 Entwicklung von JavaScript-Laufzeitumgebungen

Viele Entwickler wollten JavaScript-Code auch außerhalb von Webseiten und Webbrowsern verwenden. Dafür wurden sogenannte JavaScript-Runtimes beziehungsweise JavaScript-Laufzeitumgebungen entwickelt, welche ohne eine Webbrowser-Umgebung auskommen. [2]

Was sind Laufzeitumgebungen

JavaScript-Laufzeitumgebungen werden häufig mit JavaScript-Engines verwechselt. Eine JavaScript-Laufzeitumgebung führt im Gegensatz zur JavaScript keinen Code aus. Stattdessen stellt eine Laufzeitumgebungen die Schnittstellen zur Verfügung, um mit der Umgebung zu kommunizieren. Die bekannteste JavaScript-Laufzeitumgebung ist der Browser selbst. Der Browser als Umgebung stellt der JavaScript-Engine die Schnittstellen zur Verfügung, um beispielsweise das Dokument-Objekt-Modell (DOM) zu manipulieren oder mit dem Fenster zu kommunizieren. [11]

Für einen Webserver oder eine Datenbank wird jedoch kein Browser oder User Interface benötigt. Dafür wurden JavaScript-Laufzeitumgebungen entwickelt, welche wie ein Browser die Schnittstellen liefern. Diese Umgebungen ermöglichen es dann, beispielsweise mit dem Dateisystem oder dem Betriebssystem zu interagieren, was im Browser nicht unbedingt möglich ist. [11]

Node.js

Die erste JavaScript-Laufzeitumgebung war Node.js. Node.js wurde vom Mathematik-Doktoranden Ryan Dahl entwickelt und im Jahr 2009 veröffentlicht. [12] [9] Die Idee, die am Ende zur Entscheidung geführt hat, Node.js zu entwickeln, entstand 2005. Dahl hat einen Weg gesucht, um einen Ladebalken für das Hochladen von Dateien zu implementieren. Er versuchte den Ladebalken in 'Mongrel', einem Ruby-Webserver, zu implementieren. Die hohe Anzahl an AJAX-Anfragen während des Dateiuploads führte jedoch zu Problemen, da Ruby nur einen Thread unterstützt [12]. Daher testete er auch C und Haskell, war aber mit keiner Lösung vollständig

zufrieden. Schlussendlich entschloss sich Ryan Dahl dazu, Node.js in JavaScript zu entwickeln. Dahl begründete seine Entscheidung damit, dass JavaScript bereits eine sehr etablierte Sprache im Web ist und noch keiner die Vision von JavaScript als Programmiersprache für einen WebServer hatte. Zudem gab es bereits viele Bibliotheken, Entwickler und leistungsstarke JavaScript-Engines. [12] Im November 2009 stellte Ryan Dahl auf der JSConf EU in Berlin zum ersten Mal Node.js der Öffentlichkeit vor. Dort demonstrierte er, wie Node.js für die parallele Verarbeitung, die Erstellung eines Webservers und die Erstellung eines Chatservers benutzt werden kann. [13] [12]

Node.js ist eine Laufzeitumgebung, die auf Ereignissen basiert und nicht blockierende Ein- und Ausgabe (I/O) unterstützt. [13] Das bedeutet, dass Node.js darauf aufbaut, dass Funktionen und Codeabschnitte Ereignisse schicken, sowie auf bestimmte Ereignisse warten und darauf reagieren. Das heißt, dass in Node.js der Code nicht angehalten wird, während beispielsweise auf Ein- oder Ausgabeoperationen gewartet wird. Stattdessen reagiert der Code auf Ereignisse, wenn diese Operationen abgeschlossen sind. Das bedeutet, dass andere Teile des Codes währenddessen weiterhin ausgeführt werden können, anstatt auf das Ende einer Operation zu warten. Dies ermöglicht am Ende eine effizientere und schnellere Programmabarbeitung.

Bei Node.js wurden ungefähr 80% des Programmcodes in C/C++ und ungefähr 20% in JavaScript selber geschrieben. [13] Als JavaScript-Engine implementiert Node.js die von Google in C++ erstellte V8-Engine. [9] [13]

Deno

2018, fast 10 Jahre nach der Veröffentlichung von Node.js auf der JSConf, hat Ryan Dahl eine weitere Präsentation auf der JSConf mit dem Titel „10 Things I regret about Node.js" gehalten. [14] In dieser Präsentation berichtet er davon, welche Dinge er bei Node.js heute anders gemacht hätte. Am Ende der Präsentation stellt Ryan Dahl dann seine neu entwickelte JavaScript-Echzeitumgebung „Deno" vor. Das Ziel von Deno: Die Fehler, welche Ryan Dahl bei der Entwicklung von Node.js gemacht hat und jetzt bereut, zu beheben. [14]

Deno implementiert wie Node.js ebenfalls die V8-Engine von Google. [14] Der Teil von Deno, der nicht in JavaScript verfasst wurde, wurde für den Prototypen, statt wie in Node.js in C++, [13] in der Programmiersprache Go geschrieben. [14] Für

die Implementierung der aktuellen Version wurde sich jedoch für das Wechseln der Programmiersprache von Go auf Rust entschieden. [15]

Bun

Zum Zeitpunkt der Studienarbeit ist Bun eine sehr neue Laufzeitumgebung. Die Version 1.0 ist erst am 08. September 2023 erschienen. [16] Bun verfolgt das Ziel, besonders schnell und einfach zu sein und wirbt damit, ein All-in-one-Toolkit zu sein. Das bedeutet, dass bereits alles Notwendige direkt in Bun integriert ist. Dazu zählen unter anderem ein Package-Manager, ein Transpiler, ein Bundler und Test-Bibliotheken.[16] Außerdem ist Bun ein „Drop-in Replacement" für Node.js. Das bedeutet, dass die meisten bestehenden Node.js-Programme und -Bibliotheken direkt mit Bun funktionieren. [16]

Im Gegensatz zu Node.js und Deno verwendet Bun nicht die V8-Engine von Google, sondern die JavaScriptCore-Engine von Apple, welche auch im Safari-Browser eingesetzt wird. Bun wurde nicht in C/C++ oder Rust wie Node.js oder Deno geschrieben, sondern setzt auf die Programmiersprache Zig. Zig ist eine Low-Level-Programmiersprache, die, wie Bun, eine hohe Geschwindigkeit verspricht. [17]

2.4 Die Bedeutung von Prozessen und Threads

Ein Computer muss viele Aufgaben schnell abarbeiten. Damit die CPU alle Aufgaben rechtzeitig bewältigen kann, müssen die Aufgaben sehr effizient und damit parallel abgearbeitet werden. Ein HTTP-Webserver, der mit einer HTML-Datei aus dem Dateisystem antwortet, profitiert beispielsweise sehr von der parallelen Verarbeitung. Für die CPU ist die Zeit, die das Dateisystem zum Antworten braucht, eine halbe Ewigkeit. In der Zeit kann die CPU andere Aufgaben beziehungsweise Anfragen an den Webserver abarbeiten. [18]

Um die parallele Verarbeitung zu verstehen, ist es wichtig, das Konzept von Prozessen und Threads zu kennen.

Prozesse

Ein Prozess repräsentiert eine eigenständige Instanz eines Programms mit all seinen Parametern und Ressourcen, auch bekannt als Prozesskontext. Dieser Kontext ist

erforderlich, um das Programm auszuführen und ist unabhängig von anderen laufenden Prozessen. Das bedeutet, dass es mehrere Prozesse geben kann, welche dasselbe Programm ausführen. Jeder von diesen Prozessen besitzt jedoch eine eigene Instanz vom Programmcode, einen eigenen Adressraum im Speicher, einen eigenen Cache, einen eigenen Registerinhalt und eigene Threads. [18]

Die CPU ist dafür verantwortlich, alle laufenden Prozesse abzuarbeiten. Da mehrere Prozesse auf einmal laufen, muss die CPU diese parallel abarbeiten. Ein CPU-Kern kann aber nur an einem Prozess gleichzeitig arbeiten. Deswegen arbeitet ein CPU-Kern für einen bestimmten Zeitabschnitt an einem Prozess und wechselt dann zu einem anderen. Dadurch können mehrere Prozesse „pseudo-parallel" abgearbeitet werden. In der Praxis verfügen CPUs in der Regel über mehrere Kerne, was teilweise echte Parallelität ermöglicht. [18]

Threads

Ein Thread ist ein sequentieller Kontrollfluss innerhalb eines Prozesses. Ein Prozess kann mehrere Threads gleichzeitig ausführen, wobei mindestens ein Thread, der sogenannte Main-Thread, vorhanden sein muss. Threads teilen bestimmte Daten miteinander, wie den Adressraum oder globale Variablen, während sie auch individuelle Daten besitzen, wie einen eigenen Stack und eigene Signale. Threads bieten den Vorteil, dass sie deutlich schneller erstellt werden können als Prozesse. Daher sind sie nützlich, um die Leistung von Anwendungen durch Parallelität zu beschleunigen. [18]

2.5 Asynchrone Programmierung in JavaScript: Callbacks und Promises

JavaScript ist eine Single-Threaded Programmiersprache. Trotzdem ermöglicht JavaScript die asynchrone Programmierung durch Callbacks und ab dem ECMAScript Standard 6.0 durch „async", „await" und Promises. [19] [5]

Callbacks

Eine Funktion kann, wenn sie eine asynchrone Operation ausführt, einen speziellen Parameter definieren. Dieser Parameter ist ein Callback-Parameter beziehungsweise eine Callback-Funktion. Callback-Funktionen ermöglichen die Ausführung von Code,

während auf das Ergebnis einer asynchronen Operation gewartet wird, ohne dabei den Hauptprogrammablauf zu blockieren. Stattdessen wird der Code, der nach dieser asynchronen Operation ausgeführt werden soll, in der Callback-Funktion definiert. Sobald die asynchrone Operation abgeschlossen ist, wird die Callback-Funktion aufgerufen und der entsprechende Code ausgeführt. [4] [5]

Ein Beispiel für einen einfachen Callback ist die „setTimeout-Funktion". Diese Funktion akzeptiert zwei Parameter. Der erste Parameter ist die Callback-Funktion, welche ausgeführt werden soll, sobald die „setTimeout-Funktion" den Callback auslöst. Im Fall der „setTimeout-Funktion" wird der Callback ausgelöst, wenn die Zeit, welche im zweiten Parameter angegeben wird, abgelaufen ist. Dabei wird der Hauptprogrammablauf nicht blockiert. Das bedeutet, dass im Codebeispiel 2.1 erst „Hello World" und dann „Ich habe 1s gewartet!" in die Konsole geschrieben wird.

```
1 setTimeout(() => console.log("Ich habe 1s gewartet!"), 1000);
2 console.log("Hello World");
```

Listing 2.1: Beispiel für eine Funktion mit Callback-Parameter

Ein Nachteil von Callback-Funktion ist, dass bei der Abhängigkeit von asynchronen Operationen, wie in Codebeispiel 2.2 schnell stark verschachtelter Code entsteht. Dieses Phänomenen wird auch als „Callback-Hell" bezeichnet. [4]

```
1 fs.readFile('file1.txt', 'utf8', function(err, data1) {
2     fs.readFile('file2.txt', 'utf8', function(err, data2) {
3         fs.readFile('file3.txt', 'utf8', function(err, data3) {
4             console.log('Daten aus file1:', data1);
5             console.log('Daten aus file2:', data2);
6             console.log('Daten aus file3:', data3);
7         });
8     });
9 });
```

Listing 2.2: Vereinfachtes Beispiel: „Callback-Hell"

Async, Await & Promises

Um das Problem der „Callback-Hell" zu lösen, wurden mit dem ECMAScript Standard 6.0 im Jahr 2015 Promises eingeführt. [19] Anstatt wie bei Callbacks einen Parameter zu übergeben, gibt die Funktion mit asynchroner Operation ein Promise-Objekt zurück. Der Code, welcher die Funktion mit der asynchronen Operation aufruft, kann dann, wenn das Promise eingelöst wurde, auf die abgeschlossene Operation reagieren. Eine Möglichkeit, auf das Promise zu reagieren, ist mit „then" und „catch". Die einfachen Callbacks „then" und „catch" können in synchronem, sowie

asynchronem Code verwendet werden. Diese Vorgehenweise ist im Codebeispiel 2.3 beschrieben. [4] [5]

```
const operation = () {
    return new Promise((resolve) => {
        setTimeout(() => resolve('Operation abgeschlossen!'), 1000)
    });
}
operation()
  .then(message => console.log(message))
  .catch(error => console.error('Fehler:', error));
```

Listing 2.3: Einfaches Beispiel für die Verwendung von Promises mit „then" und „catch"

Die Verwendung von „then" und „catch" führt bei vielen voneinander abhängigen asynchronen Operation ebenfalls zur „Callback-Hell". Deswegen sollte bei vielen Abhängigkeiten der Syntax mit „async" und „await" verwendet werden. Anstatt auf den Callback zu „subscriben" und mit dem Hauptprogrammablauf fortzufahren, wird mit dem Keyword „await" gewartet, bis die asynchrone Operation abgeschlossen ist. Das „await" Keyword kann nur in einer asynchronen Funktion verwendet werden. Eine asynchrone Funktion muss mit dem Keyword „async" versehen werden. Für die Fehlerbehandlung muss dann „try" und „catch" verwendet werden. Das Codebeispiel 2.4 zeigt den „then", „catch" Codeblock aus Codebeispiel 2.3 umgeschrieben zur Vorgehensweise mit „async" und „await". [4]

```
const executeWait = async () => {
    try {
        const message = await operation();
        console.log(message);
    } catch (error) {
        console.error('Fehler:', error);
    }
};
executeWait();
```

Listing 2.4: Alternative zu „then" und „catch" mit „async" und „await"

2.6 Vergleich der Thread-Erstellung in Node.js, Deno und Bun

Node.js, Deno und Bun implementieren, wie jede Laufzeitumgebung, den ECMA-Script Standard. In diesem Standard sind die in Kapitel 2.5 beschriebenen Promises und Callbacks definiert. Daher unterstützen auch Node.js, Bun und Deno

die Asynchronität mithilfe von Promises und Callbacks. Zusätzlich implementieren diese Laufzeitumgebungen jeweils eine Worker-Thread-API, mit der zusätzlich Threads zum Hauptthread erstellt werden können. In den folgenden Kapitel wird die Erstellung von Threads in den drei betrachteten Laufzeitumgebungen beschrieben.

Node.js

In Node.js können Threads mit der „node:worker_threads" Bibliothek erstellt werden. Diese Bibliothek implementiert die Worker-Klasse. Um einen neuen Threads zu erstellen, wird mithilfe des „new"-Keywords eine Instanz von der Worker-Klasse erzeugt. Der Instanz vom Worker müssen bei der Initialisierung zwei Parameter übergeben werden. Der erste Parameter ist der Pfad zur Datei, welche der Worker ausführen soll. Um die Datei auszuführen, in welcher die Worker-Instanz erzeugt wurde, kann die „__filename" Variable übergeben werden. Außerdem kann dem Worker-Thread mit dem zweiten Parameter Daten übergeben werden, auf welche der Thread dann zugreifen kann. Mithilfe der „isMainThread"-Variable aus der „worker_threads"-Bibliothek kann geprüft werden, ob der Code von einem Worker- oder Main-Thread ausgeführt wird. Der Beispielcode 2.5 zeigt einen minimalen Aufbau, in dem ein Thread erstellt wird. Der Thread gibt dann die ihm übergebenen Daten aus. [20]

```
1 const { Worker, isMainThread, workerData } = require('node:
    worker_threads}');
2 if (isMainThread) {
3     /* This is the main thread */
4     new Worker(__filename, { workerData: 'Hello from the Worker
        Thread!' });
5 } else {
6     console.log(workerData);
7 }
```

Listing 2.5: Minimalbeispiel: Erstellen von drei Threads mit der Node.js Worker-Threads-Bibliothek

Damit mögliche Datenverluste vermieden werden, sollten die Threads sanft beendet werden. Um einen Thread zu beenden, kann die „terminate()"-Methode auf der Worker-Instanz aufgerufen werden. [20]

JavaScript ist eine single-threaded-Programmiersprache. Deshalb wird für jeden Worker-Thread eine komplett neue Instanz vom V8-Javascript-Interpreter gestartet, mit eigenen Variablen und einem eigenen Thread der den Javascript Code

ausführt. Damit die Worker-Threads untereinander kommunizieren können, implementiert Node.js ein Message-System. Der Main-Thread kann Nachrichten von den Worker-Threads mithilfe eines Callbacks in der Worker-Klasse empfangen oder senden. Ein Worker-Thread kann auf die gleiche Weise mit dem Main-Thread über die „parentPort"-Klasse kommunizieren. Es gibt dabei verschiedene Arten von Events, auf die reagiert werden kann. Dazu gehören Callbacks auf ein „message"-, ein „error"- oder ein „exit"-Event. Um eine Nachricht an den Main-Thread oder den Worker-Thread zu senden, wird die „postMessage"-Methode verwendet. Eine mögliche Implementierung ist in Codebeispiel 2.6 zu sehen. [20]

```
1 /* Empfangen von Nachrichten im Main-Thread */
2 worker.on('message', (msg) => console.log(msg));
3 worker.on('error', (err) => console.log(err));
4 worker.on('exit', (code) => console.log(code));
5
6 /* Senden von Nachrichten an den Main-Thread im Worker-Thread */
7 parentPort.postMessage('Hello from the Worker Thread!');
```

Listing 2.6: Empfangen von Messages eines Worker-Threads

Deno

Deno implementiert Worker-Threads mithilfe der „Web Worker API". [21] Der Aufbau des Codes ist sehr ähnlich zu Node.js. Die Implementierung unterscheidet sich lediglich an wenigen Stellen in der Syntax.

Für die Erstellung eines Worker-Threads muss eine Instanz der Worker-Klasse erstellt werden. Bei der Instanziierung muss der Pfad zur Datei angegeben werden, welcher der Worker-Thread ausführen soll. Zusätzlich können weitere Optionen durch den zweiten Parameter übermittelt werden. Da Deno aktuell nur Worker mit dem Typ „module" unterstützt, muss diese Option zwingend mit übergeben werden. Ähnlich wie bei Node.js implementiert Deno ein Message-System. Über „worker.postMessage" oder „self.postMessage" können Nachrichten zwischen dem Main-Thread und Worker-Thread ausgetauscht werden. Mithilfe von „onmessage" oder „onerror" auf der Worker-Klasse kann eine Funktion registriert werden, welche die Nachrichten empfängt. [22]

```
1  /* main.js */
2  const worker = new Worker(new URL("./worker.js", import.meta.url).
       href, { type: "module" });
3  worker.onmessage = (evt) => {
4    console.log(evt.data);
5  };
6
7  /* worker.js */
8  self.postMessage('Hello from the Worker Thread!');
9  self.onmessage = (evt) => {
10   console.log(evt.data);
11 };
12 self.close();
```

Listing 2.7: Erstellen eines Worker-Threads in Deno

Bun

Da Bun ein „Drop-in Replacement" für Node.js ist, [16] funktioniert das Codebeispiel 2.5 mit der „node:worker_threads"-Bibliothek aus Node.js direkt auch mit Bun. Zusätzlich existiert in Bun auch eine eigene Implementierung der Worker-Threads. Dafür verwendet Bun, wie Deno auch, die „Web Worker API". [21] [23] Deswegen funktioniert das Codebeispiel 2.7 in Deno ebenfalls in Bun. Im Gegensatz zu Deno unterstützt Bun „CommonJs" und „EsModules" in Worker-Threads. Dadurch muss bei der Instanziierung in Bun die Option „type: module" nicht übergeben werden. [23]

3 Untersuchung des Thread-Managements und Leistungsvergleich von Node.js, Deno und Bun

Im folgenden Kapitel erfolgt eine Analyse des Thread-Managements der einzelnen Laufzeitumgebungen mithilfe eines synthetischen Benchmarks.

3.1 Testumgebung

Die Benchmark-Durchführung wird auf einem System mit Debian 12 Linux durchgeführt. Dieses System ist mit einem 10-Kern-Prozessor und 16 logischen Threads ausgestattet. Der verfügbare Arbeitsspeicher beträgt 32 GB RAM. Eine detaillierte Auflistung der genauen Spezifikationen ist nachfolgend aufgeführt:

Betriebssystem:	Debian GNU/Linux 12 (bookworm) x86_64
CPU:	i5 12600KF, 3.70GHz, (6+4) Kerne, 16 Threads
Arbeitsspeicher:	32 GB DDR4-3600 DIMM CL18
Speicher:	2TB PCIE 4.0 M.2 SSD

Die Laufzeitumgebungen werden in der zum Zeitpunkt des Benchmarks aktuellsten (LTS) Version verwendet. Zum Stand vom 18.01.2023 waren dies:

Node.js:	v20.11.0
Deno:	1.40.2
Bun:	1.0.23

3.2 Ablauf der Untersuchung

Der synthetische Benchmark soll die parallele Verarbeitung der drei Laufzeitumgebungen testen. Dabei wird die Laufzeit, CPU-Auslastung und maximale Arbeitsspei-

chernutzung gemessen. Zur Messung wird ein separates „time"-Programm mit dem Befehl „/usr/bin/time -v <Befehl>" verwendet. Der Befehl gibt die Zeit in den drei Kategorien „real", „user" und „sys" an. Um die Laufzeit des Prozesses zu messen wird die angegebene „real"-Zeit verwendet. Diese gibt die reale Zeit von Beginn bis Abschluss des Prozesses an. Um die CPU Auslastung zu messen, können die „user"- und „sys"-Zeiten addiert werden. Die „user"-Zeit gibt an, wie viel Zeit der Prozess und seine Threads im sogenannten User-Raum der CPU verbracht haben. Die „sys"-Zeit dagegen gibt die verbrauchte Zeit im Kernel-Modus der CPU an. Es ist entscheidend zu beachten, dass die Zeit im User- und Kernel-Raum aufgrund von Multithreading möglicherweise länger ist als die in der realen Zeit gemessene Gesamtlaufzeit. Dieses Verhalten wird im folgenden Benchmark durch die Erstellung mehrerer Threads definitiv zu beobachten sein. Für die Messung der maximalen Belegung des Arbeitsspeichers wird der unter „Maximum resident set size" angegebene Wert verwendet.

Um die Parallele Verarbeitung optimal testen zu können, sollten die Threads eine möglichst rechenintensive Aufgabe bearbeiten. Dafür eignet sich eine Aufgabe, welche eine rekursive Funktion benötigt. Für diesen Benchmark wird in den einzelnen Threads die Fibonacci-Reihenfolge berechnet. Die Fibonacci-Reihe ist eine Zahlenfolge, bei der jede Zahl die Summe der beiden vorherigen Zahlen ist. Die verwendete Implementierung für den Benchmark ist im Codeausschnitt 3.1 zu sehen.

```
1 function fibonacci(n, memo = {}) {
2     if (n <= 1) return n;
3     if (memo[n]) return memo[n];
4
5     memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
6     return memo[n];
7 }
```

Listing 3.1: Berechnung der Fibonacci Reihenfolge

Es werden Messwerte für verschiedene Thread-Konfigurationen erfasst, darunter 1 Thread (Grundverbrauch), 8 Threads (die Hälfte der logischen Threads in der Testumgebung), 16 Threads (alle logischen Threads), 32 Threads (das Doppelte der logischen Threads) und 64 Threads (das Vierfache der logischen Threads). Jeder dieser Thread-Konfigurationen wird für jede Laufzeitumgebungen durchgeführt und zehn mal wiederholt. Aus den 10 Versuchen wird im Anschluss für jede Thread-Konfiguration der arithmetische Mittelwert berechnet. Der vollständige Code des Benchmarks wurde auf dieses GitHub Repository hochgeladen: https://github.com/Fabianofski/Studentproject-II

3.3 Hypothesen

Bun verspricht eine besonders hohe Geschwindigkeit und sollte die Threads am schnellsten abarbeiten können. Node.js und Deno verwenden die selbe JavaScript-Engine und weisen eine sehr ähnliche Struktur auf, was darauf hindeutet, dass ihre Ergebnisse relativ ähnlich sein sollten. Allerdings verwendet Deno, sowie Bun die Web Workers API, während Node.js eine eigenständige API einsetzt.

Aufgrund der unterschiedlichen Worker-Threads APIs und JavaScript-Engines kann erwartet werden, dass sich in den Ergebnissen deutliche Unterschiede zwischen allen Laufzeitumgebungen zeigen werden. Dabei kann angenommen werden, dass Deno und Bun aufgrund modernerer Technologien ressourcensparender und schneller sein sollten.

3.4 Durchführung

Laufzeit

Bun weist die schnellste Laufzeit auf und ist doppelt so schnell wie Node.js und fast dreimal so schnell wie Deno. Überraschenderweise ist Deno deutlich am langsamsten. Der Anstieg der Laufzeit bei Deno ist außerdem nahezu linear.

Bei Bun und Node.js ist ein deutlicher Anstieg ab 32 Threads erkennbar. Der Anstieg ist damit zu erklären, dass die Testumgebung nur 16 logische Threads zur Verfügung hat und dadurch ab 32 Threads nicht mehr alle Threads auf einmal abgearbeitet werden können.

Threads	Laufzeit
1	0.03s
8	0.04s
16	0.09s
32	0.24s
64	0.5s

(a) Node.js

Threads	Laufzeit
1	0.03s
8	0.11s
16	0.19s
32	0.37s
64	0.73s

(b) Deno

Threads	Laufzeit
1	0.01s
8	0.02s
16	0.03s
32	0.12s
64	0.21s

(c) Bun

Abbildung 1: Messdaten der Laufzeit in Abhängigkeit von der Anzahl der Threads.

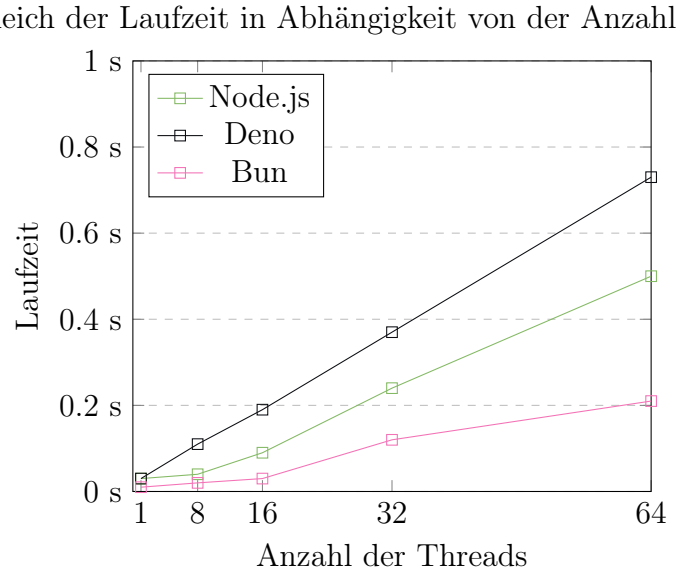

Abbildung 2: Vergleich der Laufzeiten für Node.js, Deno und Bun.

CPU-Auslastung

Laut den gemessenen Werten belastet Bun bis 16 Threads die CPU am wenigsten. Deno belastet die CPU bis 16 Threads etwas mehr als Bun, ab 32 Threads aber deutlich weniger als Bun. Auffällig ist hier, dass Denos Messwerte, wie bei der Laufzeit, ebenfalls linear ansteigen. Node.js belastet die CPU am meisten. Die Last ist bei bei 64 Threads über 4-mal so hoch wie bei Deno und fast 3-mal so hoch wie bei Bun.

Da beim Erstellen mehrerer Threads die CPU an mehreren Aufgaben parallel arbeiten kann, ist die CPU Auslastung höher als die tatsächliche Laufzeit. Unter Verwendung von 64 Threads zeigt sich eine CPU-Auslastung bei Node.js, die etwa 8-mal höher ist als die tatsächliche Laufzeit. Im Fall von Deno ist die Auslastung um das 1,09-fache erhöht, während bei Bun die CPU-Auslastung etwa 6,66-mal höher liegt als die Laufzeit.

Threads	CPU Last
1	0.03s
8	0.28s
16	0.73s
32	1.9s
64	4s

(a) Node.js

Threads	CPU Last
1	0.02s
8	0.11s
16	0.21s
32	0.4s
64	0.8s

(b) Deno

Threads	CPU Last
1	0.01s
8	0.09s
16	0.2s
32	0.78s
64	1.4s

(c) Bun

Abbildung 3: Messdaten der CPU Auslastung in Abhängigkeit von der Anzahl der Threads.

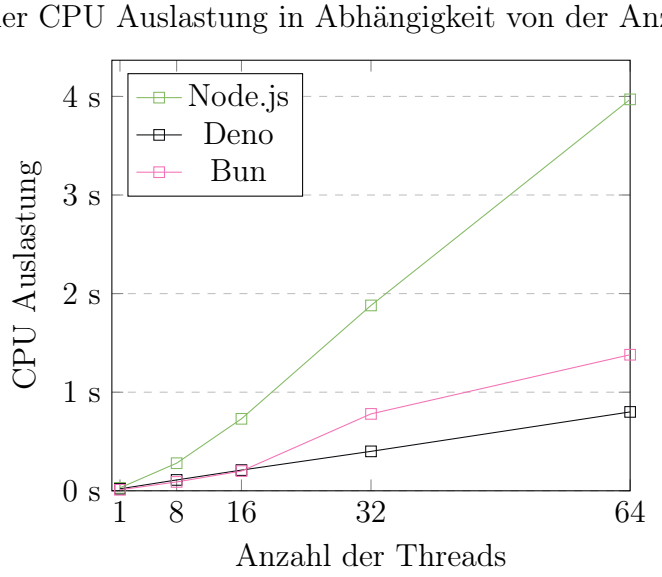

Abbildung 4: Vergleich der CPU Auslastungen für Node.js, Deno und Bun.

Arbeitsspeicherauslastung

Der Grundverbrauch (1 Thread) von Node.js beträgt 54 kB, während Deno 63 kB und Bun mit 70 kB den höchsten Verbrauch aufweisen. Mit zunehmender Anzahl von Threads benötigt Node.js den größten Arbeitsspeicher. Im Vergleich dazu zeigt Bun eine niedrigere Steigerung und erfordert bei mehr Threads weniger Arbeitsspeicher als Node.js. Bei 64 Threads verbraucht Bun 260kB und Node.js das 1,7-fache mit 450kB. Bei Deno bleibt die Arbeitsspeichernutzung nahezu konstant. Der Verbrauch steigt lediglich um ein paar kB mit zunehmender Anzahl von Threads. Unter Verwendung von 64 Threads verbraucht Node.js das 6,33-fache und Bun das 3,66-fache von Deno.

Threads	RAM Last
1	54,000kb
8	110,000kb
16	180,000kb
32	210,000kb
64	450,000kb

(a) Node.js

Threads	RAM Last
1	63,000kb
8	66,000kb
16	67,000kb
32	69,000kb
64	71,000kb

(b) Deno

Threads	RAM Last
1	70,000kb
8	110,000kb
16	130,000kb
32	180,000kb
64	260,000kb

(c) Bun

Abbildung 5: Messdaten der RAM Auslastung in Abhängigkeit von der Anzahl der Threads.

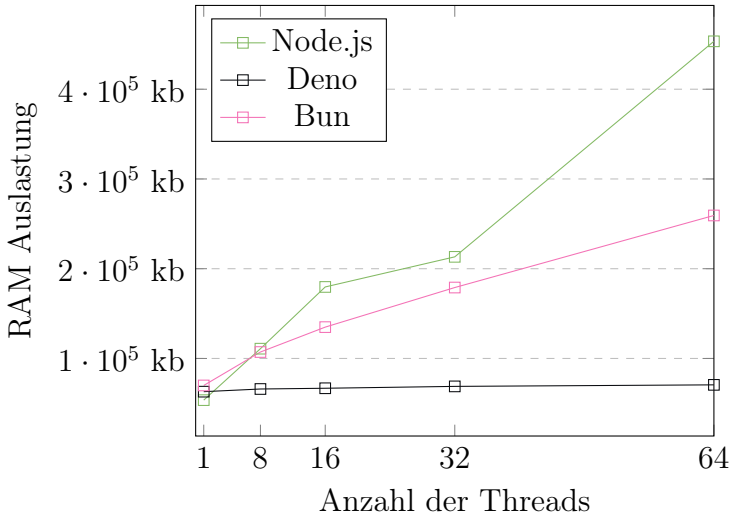

Abbildung 6: Vergleich der RAM Auslastungen für Node.js, Deno und Bun.

3.5 Auswertung der Ergebnisse

Bei Verwendung mehrerer Threads sollte man beobachten können, dass die Laufzeit bis zur Erreichung des logischen Thread-Limits des Testsystems nur minimal ansteigt, bevor sie dann deutlich ansteigt. Darüber hinaus sollte die CPU-Auslastung im Verhältnis zur Laufzeit erheblich höher sein. Mit zunehmender Anzahl von Threads sollte auch der Speicherverbrauch steigen. Bei Deno zeigt sich jedoch ein lineares Ansteigen der Laufzeit, während die CPU-Auslastung lediglich dem 1,09-fachen der Laufzeit entspricht und der Speicherverbrauch konstant bleibt. Basierend auf diesen Beobachtungen scheint es einen Fehler im Thread-Management von Deno (v1.40.2) zu geben. Daher ist ein direkter Vergleich zwischen Deno und Node.js oder Bun nicht möglich.

In den Messwerten ist klar erkennbar, dass Bun die Leistungsversprechen einhalten kann. Im Vergleich zu Node.js ist Bun deutlich schneller, belastet weniger die CPU und benötigt auch weniger Arbeitsspeicher. Dass Node.js langsam und ressourcenfressend ist, resultiert aus den Entwicklungsschwerpunkten. Da Node.js lange Zeit die einzige ernsthafte Laufzeitumgebung auf dem Markt war, lag der Fokus darauf, neue Funktionen zu implementieren, anstatt die Ressourcennutzung und die Laufzeit zu optimieren. Mit der Einführung von Deno und Bun könnte sich dieser Fokus jedoch ändern und es besteht die Möglichkeit, dass sich die Ressourcennutzung und Laufzeit von Node.js in der Zukunft verbessert.

4 Fazit und Ausblick

JavaScript hat in den vergangenen Jahren eine beeindruckende Entwicklung durchlaufen. Von der ersten einfachen JavaScript-Engine bis hin zu den heutigen modernen und optimierten Engines mit Just-in-Time-Compilation hat sich die Landschaft rund um JavaScript stark verändert. Ständig werden neue Technologien und Bibliotheken entwickelt, um das Maximum an Performance aus JavaScript herauszuholen.

Wie der Benchmark zeigt hat Bun sich als die schnellste Laufzeitumgebung erwiesen. Trotzdem ist Bun, auch wenn bereits Version 1.0 veröffentlicht wurde, genau wie Deno, als relativ neues Projekt möglicherweise noch nicht vollständig produktionsreif. Aktuell empfiehlt es sich weiterhin, für Produktionssysteme auf das langjährig bewährte Node.js zurückzugreifen, selbst wenn dies mit einem Verlust an Laufzeit und Ressourcen einhergeht.

Die Einführung von Bun und Deno und die damit verbundene Konkurrenz haben Node.js unter Druck gesetzt. Die Herausforderung in Bezug auf Ressourcen und Performance wird Node.js dazu zwingen, die Plattform zu optimieren und nachzubessern, um mit den neuen Laufzeitumgebungen Schritt zu halten. Deno und Bun werden in der Zukunft ebenfalls stetig weiterentwickelt und werden noch weiter optimiert sowie mit neuen Features ausgestattet werden. Dies wird zu einer weiteren Verbesserung der Ressourcennutzung und Laufzeitumgebungen führen, was letztendlich sowohl Entwicklern als auch Endbenutzern zugutekommen wird.

Literaturverzeichnis

[1] Stack Overflow. Stack Overflow Developer Survey, 2023.

[2] Allen Wirfs-Brock and Brendan Eich. JavaScript: the first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):6–11, 2020.

[3] Marc Andreessen. Innovators of the net: Brendan Eich and JavaScript, 1998. http://web.archive.org/web/20080208124612/http:/wp.netscape.com/comprod/columns/techvision/innovators_be.html; abgerufen am 18.07.2023.

[4] Marijn Haverbeke. *Eloquent JavaScript*. No Starch Press, 2018.

[5] ECMA. ECMAScript® 2023 Language Specification, 2023. https://262.ecma-international.org/14.0/; abgerufen am 06.02.2024.

[6] Rainer Hahnekamp. JavaScript Engines: Performance-Steigerung der Browser. *heise online*, 08.01.2019. Abgerufen am 01.11.2023.

[7] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. New York, USA, 2022. Association for Computing Machinery.

[8] James Gray. Google Chrome: the making of a cross-platform browser. *Linux J.*, 2009.

[9] Mathias Bynens. Celebrating 10 years of V8 · V8, 2018. https://v8.dev/blog/10-years; abgerufen am 01.11.2023.

[10] V8 Developer Team. V8 JavaScript Engine Documentation, 2023. https://v8.dev/docs; abgerufen am 18.07.2023.

[11] Oliver Ochs. *JavaScript für Enterprise-Entwickler: Professionell programmieren im Browser und auf dem Server*. dpunkt-Verlag, Heidelberg, 1. aufl. edition, 2012.

[12] Sebastian Springer. *Node. js: The Comprehensive Guide*. Rheinwerk Publishing Inc, Boston, 2022.

[13] Ryan Dahl. Original Node.js presentation - JSConf EU, 2009.

[14] Ryan Dahl. 10 Things I Regret About Node.js - JSConf EU, 2018.

[15] Deno. GitHub-Repository: Deno. https://github.com/denoland/deno; abgerufen am 06.02.2024.

[16] Bun. Bun 1.0, 2023. https://bun.sh/blog/bun-v1.0; abgerufen am 06.02.2024.

[17] Timo Zander and Silke Hahn. Framework Bun: "Schneller als mir irgendwer glaubt" – Jarred Sumner im Interview. *heise online*, 2022.

[18] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems. 4th Edition.* Pearson Education, 2015.

[19] ECMA. ECMAScript® 2015 Language Specification, 2015. https://262.ecma-international.org/6.0/; abgerufen am 06.02.2024.

[20] Node.js. Worker threads | Node.js v21.6.1 Documentation. https://nodejs.org/api/worker_threads.html; abgerufen am 06.02.2024.

[21] MDN Web Docs. Using Web Workers - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers; abgerufen am 06.02.2024.

[22] Deno. Workers | Deno Docs. https://docs.deno.com/runtime/manual/runtime/workers; abgerufen am 06.02.2024.

[23] Bun. Workers – API | Bun Docs. https://bun.sh/docs/api/workers; abgerufen am 06.02.2024.

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich:

1. dass ich meine Studienarbeit selbstständig verfasst habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe,

3. dass ich die Studienarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Berlin, 19.02.2024

Ort, Datum	Fabian Friedrich