Der Weg zu performantem NodeJS – Do’s and Dont‘s

Projektarbeit Nr.:	[redacted]
vorgelegt am:	[redacted]
von:	[redacted]
Matrikelnummer:	[redacted]
Berufsakademie:	[redacted]
Studienbereich:	Praktische Informatik
Studiengang:	B. Eng. Praktische Informatik
Kurs:	[redacted]
Ausbildungsstätte:	dotSource GmbH Goethestraße 1 07743 Jena
Betreuer Praxisbetrieb:	[redacted]
Gutachter Berufsakademie:	[redacted]

Head Office Jena
Goethestraße 1
07743 Jena
FON +49 (0) 3641 797 9000
FAX +49 (0) 3641 797 9099
E-MAIL info@dotSource.de

Office Berlin
Pappelallee 78/79
10437 Berlin
FON +49 (0) 30 220 122 360

Office Leipzig
Hainstraße 1–3
04109 Leipzig
FON +49 (0) 341 9919 1000

Bankverbindung
Deutsche Bank Jena
IBAN DE63 8207 0000 0633 7778 00
BIC DEUTDE8EXXX

Commerzbank Jena
IBAN DE31 8204 0000 0259 9934 00
BIC COBADEFF821

Sparkasse Jena
IBAN DE35 8305 3030 0018 0037 61
BIC HELADEF1JEN

Geschäftsführer
Christian Otto Grötsch
Christian Malik
Frank Ertel

Gerichtsstand
Amtsgericht Jena
HRB 210634
USt-IdNr.: DE246243309
Steuer-Nr.: 162/107/03164

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH

I Inhaltsverzeichnis

II Abbildungsverzeichnis

1 Was ist NodeJS?

JavaScript ist in der heutigen Zeit eine der meist genutzten Programmiersprachen überhaupt. Mit dem Wachstum des Internets und der immer komplexeren Funktionalität von Webseiten ist JavaScript aus der Sicht eines Webentwicklers nicht mehr wegzudenken. Jeder dynamische Inhalt einer Webseite wird mit JavaScript umgesetzt. Beim Aufruf einer Seite werden neben den HTML und CSS-Dateien auch dazugehörige JavaScript-Dateien geladen und vom Browser und dessen JavaScript-Engine ausgeführt. Dabei wurde JavaScript so beliebt, dass es nun auch im Desktop- und Serverbereich genutzt werden sollte.

Im Jahr 2009 entwickelte Ryan Dahl NodeJS, um einen Webserver bereitzustellen, welcher viele parallele Anfragen gleichzeitig verarbeiten kann, da er mit der Performance von einem Apache-Webserver diesbezüglich unzufrieden war. Als Basis wählte er JavaScript aufgrund des besonderen ereignisgetriebenen Systems um asynchron Anfragen abarbeiten zu können.[1] Dazu wurde die JavaScript-Engine namens „V8“ aus dem Google Chrome Browser genutzt um JavaScript außerhalb des Browsers aufzuführen. Die damit einhergehenden technischen Anpassungen bringen nun auch einige Änderungen mit sich, die es zu beachten gilt, wenn man NodeJS anstelle von reinem JavaScript im Browser programmiert. Bei NodeJS handelt es sich also nicht um eine Programmiersprache, sondern vielmehr um ein Framework, dass es ermöglicht JavaScript oder eine JavaScript-Engine auf dem Desktop oder einem Server alleinstehend auszuführen. Dabei wird JavaScript allerdings um einige spezifische Funktionen von NodeJS erweitert um auf Ressourcen in der Umgebung, wie z.B. das Dateisystem zuzugreifen. Bei JavaScript im Browser sind solche Funktionen nicht vorhanden oder nur sehr schwer zu implementieren.

[1] [Aug20].

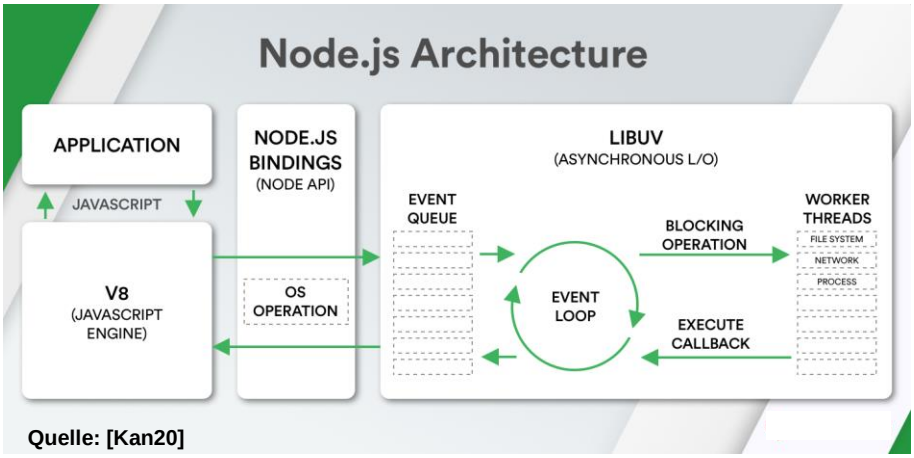
Da NodeJS sehr häufig im Bereich von Webserveranwendungen genutzt wird, ist es äußert wichtig, dass möglichst viele Anfragen gleichzeitig verarbeitet werden können ohne, dass diese sich gegenseitig blockieren. Jeder Nutzer einer Webseite möchte seine Ergebnisse schließlich in einer angemessenen Zeit sehen.
Aus der Sicht eines JavaScript-Entwicklers ist es wichtig zu verstehen wie NodeJS im Inneren funktioniert, um möglichst effizienten Programmcode zu schreiben und dieser Anforderung gerecht zu werden.
Deshalb wird in den folgenden Kapiteln zunächst die Funktionsweise von NodeJS unter Berücksichtigung des „Eventloops“, „libUV“ und „V8“ erklärt. Daraus ergeben sich einige Vorgehensweisen, die es zu beachten gilt wenn man asynchronen Programmcode entwickeln möchte. Des Weiteren sollen Best Practices gezeigt werden, die in den meisten Fällen immer einfach anzuwenden sind um eine NodeJS Anwendung noch performanter zu gestalten. Anhand verschiedener Testszenarien soll geklärt werden, welche Auswirkungen diese Methoden auf die Performance haben und warum einigen Techniken vielleicht vermieden werden sollten.
Das Ziel dieser Arbeit ist es also, den Aufbau von NodeJS zu analysieren und die wichtigsten Punkte bezüglich der Performance solcher Anwendungen herauszuarbeiten. Dazu werden Grundlegende Kenntnisse der Sprache JavaScript vorausgesetzt. Weiterhin sollen Möglichkeiten zur Optimierung von NodeJS-Anwendungen zusammengetragen und analysiert werden um diese dann ggf. in Produktivsystem einzusetzen.

2 Das Innere von NodeJS

Im folgenden Kapitel soll geklärt werden aus welchen Komponenten eine NodeJS-Umgebung zusammengesetzt ist und wie diese miteinander interagieren.

NodeJS besteht im Wesentlichen aus zwei Komponenten. Einer JavaScript-Engine und einer Bibliothek für asynchrone Ein-/Ausgabe-Verarbeitung. In der Anfangszeit von NodeJS wurde Googles JavaScript-Engine V8 genutzt. Diese war bereits im Google Chrome Browser im Einsatz und konnte somit auch in NodeJS sofort eingesetzt werden. Es existieren neben V8 auch noch anderen JavaScript-Engines, welche mittels einiger Modifikationen ebenfalls in NodeJS genutzt werden können. Chakara ist die Engine von Microsoft und kommt in deren Edge Browser zum Einsatz. In Mozilla Firefox wird eine Engine namens SpiderMonkey genutzt um JavaScript auszuführen.[2] In dieser Arbeit wird allerdings von NodeJS in der Standardkonfiguration ausgegangen, welche V8 nutzt. Um V8 auszuführen werden neben einer Schnittstelle zum Betriebssystem auch einige Ressourcen des Systems benötigt. Um außerdem noch einen asynchronen Programmablauf zu gewährleisten kommt als zweite größere Komponente der NodeJS-Welt die Bibliothek libUV zum Einsatz. Sie stellt einen vollfunktionsfähigen Eventloop bereit, einen Threadpool um synchrone Prozesse auszulagern und implementiert einige Schnittstellen zum Betriebssystem.[3]

Abbildung 1: NodeJS Architektur - Übersicht



Quelle: [Kan20]

[2] [San19a].
[3] [Ope21b].

2.1 Googles JavaScript Engine

V8 ist die JavaScript-Engine welche im Google Chrome Browser JavaScript ausführt. Web Schnittstellen oder das DOM wird von Browser selbst bereitgestellt. Dabei ist V8 unabhängig vom Browser und kann alleinstehend ausgeführt werden.[4] Sie ist opensource und größtenteils in C++ programmiert und somit kann als Bibliothek in jedem C++-Programm eingebunden und genutzt werden.[5]

V8 übersetzt den JavaScript-Quelltext zur Laufzeit erst in Bytecode und dann in Maschinencode.[6] In anderen Sprachen wie C++ wird der Quelltext bereits vor der Ausführung von einem Compiler entweder direkt in Maschinencode übersetzt oder wie in Java erst in Bytecode umgewandelt bevor dieser dann von der Java Laufzeitumgebung in Maschinencode übersetzt wird. Dabei haben die Compiler dieser statisch typisierten Sprachen mehrere Vorteile bezüglich der Performance des Maschinencodes der am Ende dabei entsteht. [7] Beispielweise ist zu jeder Zeit bekannt von welchem Datentyp eine Variable ist und wie die Signaturen von Funktionen dahingehend aussehen. Der Compiler kann mit diesen Informationen beim Übersetzen des Quelltextes in Maschinencode Optimierungen vornehmen, sodass der bereits optimierte Code später ausgeführt werden kann.

V8 kann auf dieser Basis aus zwei Gründen keine Optimierungen vornehmen. Zum einen ist JavaScript dynamisch typisiert.[8] Zum anderen wird der Quelltext erst zur Laufzeit in Maschinencode übersetzt. Es ist also nicht bekannt welche Variable von welchem Typ ist, da sich dieser Typ stets ändern kann, während das Programm ausgeführt wird. Doch V8 ist trotzdem in der Lage JavaScript-Quelltext mittels anderer Techniken stark zu optimieren. Auch wenn JavaScript nicht für sehr rechenaufwendige Aufgaben ausgelegt ist, gibt es doch Abschnitte im Programm welche einige Optimierungen benötigen. Intern enthält V8 zwei verschiedene Compiler. Der erste Compiler wird „Ignition“ genannt und interpretiert den JavaScript-Quelltext als

[4] [Ope20].
[5] [Goo21].
[6] [Wan19].
[7] Vgl. ebenda.
[8] [Moz21].

Bytecode. [9] Er erfüllt dabei mehrere Aufgaben um den Quelltext zu verbessern. Im Gegensatz zu vollwertigen Compilern übersetzen Interpreter jede Zeile des Quelltextes einzeln und führen diese aus. Ihnen fehlt also die „Sicht“ auf das ganze Programm.[10] Also merkt sich Ingnition welche Zeilen interpretiert wurden um diese gegebenenfalls wieder zu verwenden. So kann Ingnition duplizierten oder nicht verwendeten Code entfernen, um kleineren Bytecode zu produzieren und somit Speicherplatz zu sparen, da solche „just in time“ Compiler einen sehr großen „memory overhead“ mit sich bringen.[11] Um zu verhindern das Funktionen jedes Mal neu interpretiert werden müssen, sollten sie mehrfach ausgeführt werden, sammelt Ignition außerdem zur Laufzeit Informationen darüber, welche Funktionen häufig durchlaufen werden und gibt diese Informationen ebenfalls an einen Profiler weiter. Solche Abschnitte werden als „hot functions“ bezeichnet.[12] Mittels der von Ingnition gesammelten Informationen versucht der zweite Compiler namens „Turbofan“ Annahmen über die Verwendung dieser Codeabschnitte zutreffen und diesen Bytecode durch sehr stark optimierten, auf die Plattform angepassten Maschinencode zu ersetzen.[13] Hierbei muss man immer im Blick behalten, dass all dies zur Laufzeit geschieht und diese Annahmen im späteren Programmablauf nicht mehr zutreffen können. Hat Turbofan alle Information über eine Funktion, wie zum Beispiel den Datentyp der Parameter oder den Datentyp des Rückgabewertes, kann diese Funktion für diese Typen optimiert werden, um beispielweise String-Operationen oder Arrayzugriffe auf Basis des genutzten Befehlssatzes des Prozessors effizienter zu gestalten. Die nicht optimierte Variante der Funktion wird direkt durch die optimierte Variante auf dem Callstack ausgetauscht. Dieses Verfahren wird „on-stack replacement“ genannt.[14] Dabei gibt es jedoch das Problem, dass sich dieser Typ zur Laufzeit ändern kann, da JavaScript dynamisch typisiert ist. Die optimierte Funktion

[9] [Wan19].
[10] Vgl. ebenda.
[11] Vgl. ebenda.
[12] [Rad20].
[13] [Wan19].
[14] [San20].

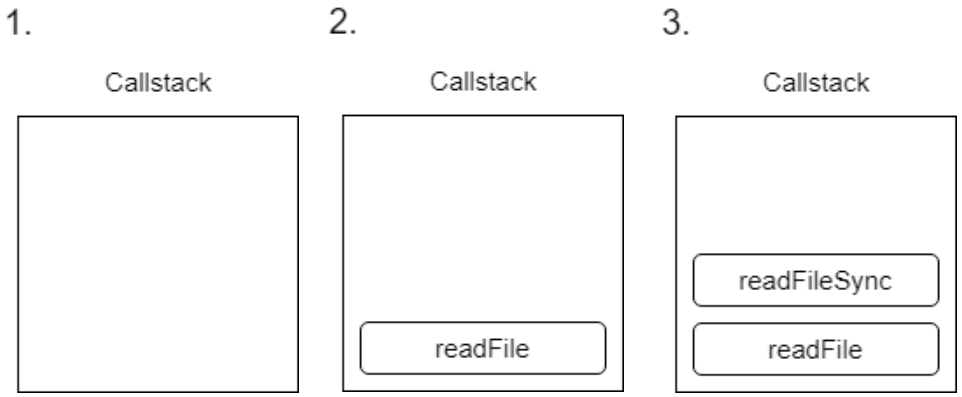
kann in diesem Fall nicht mehr genutzt werden. Sie wird dann zur Deoptimierung markiert und der optimierte Maschinencode wird einfach durch den alten Bytecode ersetzt um sie wieder für alle Datentypen verwenden zu können.[15]
Hierbei wird auch deutlich, dass der Callstack ebenfalls in V8 implementiert ist und somit auch dort überwacht und verwaltet wird. Dieser funktioniert, wie für einen Stack üblich, nach dem „last in, first out“ Prinzip. Folgendes Beispiel soll die Funktionsweise des Callstacks genauer erklären.

Abbildung 2: Callstack - Codebeispiel

```
1   const filesystem = require("fs")
2
3   function readFile(path){
4       const data = filesystem.readFileSync(path)
5       console.log(data);
6   }
7
8
9   readFile("hallowelt.txt")
10  console.log("Programm beendet!")
```

In diesem Beispiel soll ein Text von dem Dateisystem aus der „hallowelt.txt“ Datei eingelesen und auf die Konsole ausgegeben werden. Der Callstack ist dabei zu Beginn leer.

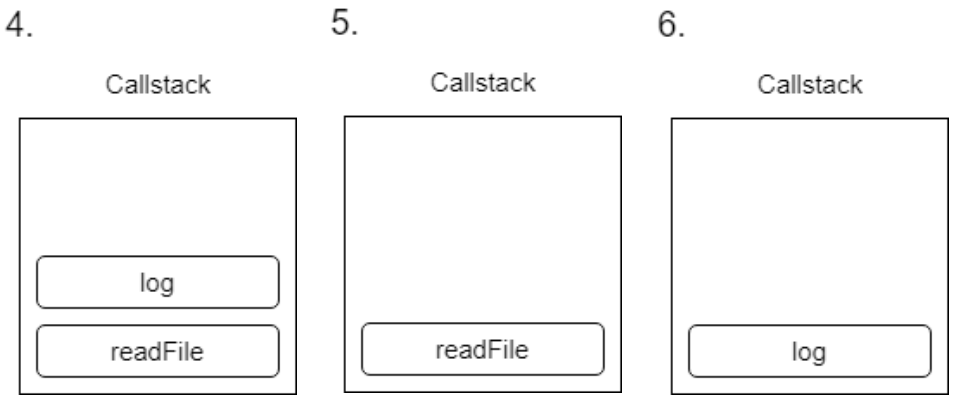
Abbildung 3: Callstack - Ablauf 1



[15] [Wan19].

Der Interpreter trifft zuerst auf Zeile 9 und die Funktion *readFile* wird auf den Callstack gelegt und anschließend ausgeführt. Da diese wiederum weitere Funktionen enthält muss dafür im dritten Schritt die Funktion *readFileSync* auf den Callstack gelegt werden. Zur Vereinfachung wird angenommen, dass diese Funktion intern aus keinen weiteren Funktionsaufrufen besteht und ohne Weiteres zurückkehrt. Sie kann dann vom Callstack entfernt werden, sodass sich nur noch *readFile* auf dem Stack befindet.

Abbildung 4: Callstack - Ablauf 2



Im vierten Schritt wird die Funktion *console.log* auf den Stack gelegt, um den Text auf der Konsole auszugeben. Nachdem das geschehen ist wird auch diese Funktion vom Stack entfernt und *readFile* befindet sich wieder als einzige Funktion auf dem Stack. Da nun innerhalb von *readFile* keine weiteren Funktionsaufrufe vorgesehen sind wird sich auch vom Stack entfernt, nach dem sie zurückgekehrt ist. Als letztes wird nochmal *console.log* auf den Stack gelegt, ausgeführt und dann wieder entfernt. Nachdem alle Funktionen ausgeführt wurden und der Stack leer ist, wird das Programm terminiert. Hierbei ist zu beachten, dass die Funktion in Zeile 4 die Datei synchron vom Dateisystem einliest. Bei einem langsamen Dateisystem kann das dazu führen, dass das Programm für eine lange Zeit auf die Antwort vom Dateisystem warten muss und während dessen blockiert ist, da V8 nur einen einzigen Callstack implementiert. Deshalb wird JavaScript auch als „single-threaded“ bezeichnet. [16] Wird solche Logik z.B. in einem Webserver verwendet, kann er in dieser Zeit keine weiteren Anfragen

[16] [Sid20].

verarbeiten, weil der einzige Thread blockiert ist. Wie der Callstack im Zusammenhang mit dem Eventloop funktioniert und damit für einen nicht blockierenden asynchronen Programmablauf sorgt wird im Kapitel 2.1 genauer erläutert.
Insgesamt gesehen verwendet V8 jedoch mehrere Threads um die Performance der Laufzeitkompilierung zu erhöhen. Es gibt beispielsweise neben dem Hauptthread um den Code auszuführen einen Thread der nur für die Optimierung zuständig ist. Ein weiterer Thread übernimmt das gesamte Profiling und weitere Threads übernehmen die Garbage-Collection.[17]

[17] [San19c].

2.2 Der Eventloop

Die opensource Bibliothek „libUV“ ist neben der JavaScript-Engine die wichtigste Komponente von NodeJS. Sie ist komplett in C implementiert und sorgt für eine nicht blockierende Ein-/Ausgabe-Verarbeitung. Sie ist dazu noch plattformunabhängig und kann V8 somit dabei helfen JavaScript auf jedem System auszuführen. [18] Sie stellt einen vollfunktionsfähigen Eventloop bereit, einen Threadpool um synchrone Prozesse auszulagern und implementiert einige Schnittstellen zum Betriebssystem.[19] LibUV stellt dazu mehrere Schnittstellen bereit um etwa auf das Dateisystem zu zugreifen oder Netzwerkkomponenten zu nutzen. Außerdem implementiert sie einen Threadpool um synchrone Prozesse auszulagern und übernimmt die Interprozesskommunikation.[20] Das wichtigste ist jedoch der Eventloop, der dafür sorgt, dass JavaScript als nicht blockierende Programmiersprache angesehen werden kann, obwohl sie nur einen Callstack besitzt und somit keine Parallelität erlaubt. Der Eventloop ist der Grund, warum NodeJS bestens für Webanwendungen geeignet ist. So ist es dadurch etwa möglich mehrere http-Anfragen entgegenzunehmen, selbst wenn eine Anfrage besonders viel Rechenzeit in Anspruch nimmt oder auf externe Ressourcen, wie z.B. einem komplexen Datenbankzugriff, warten muss.
Der Eventloop gibt den Programmablauf maßgeblich vor, indem er die Reihenfolge der Funktionsaufrufe auf Basis von Ereignissen verwaltet.

[18] [Mir21].
[19] [Ope21b].
[20] [Hax20].

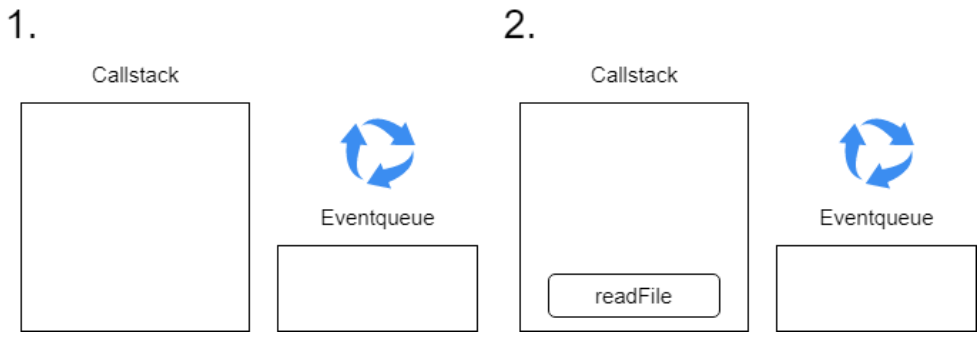
Nachdem in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** geklärt w urde wie ein Callstack funktioniert soll nun folgendes Beispiel erläutern, wie Funktionen im Zusammenhang mit dem Eventloop und dem Callstack asynchron verarbeitet werden können. Dazu wird das Beispiel leicht verändert und die synchrone Funktion durch eine asynchrone ersetzt.

Abbildung 5: Eventloop - Codebeispiel

```
1   const filesystem = require("fs")
2
3   function readFile(path){
4       filesystem.readFile(path, (err, data) => {
5           console.log(data);
6       })
7   }
8
9
10  readFile("hallowelt.txt")
11  console.log("Programm beendet!")
```

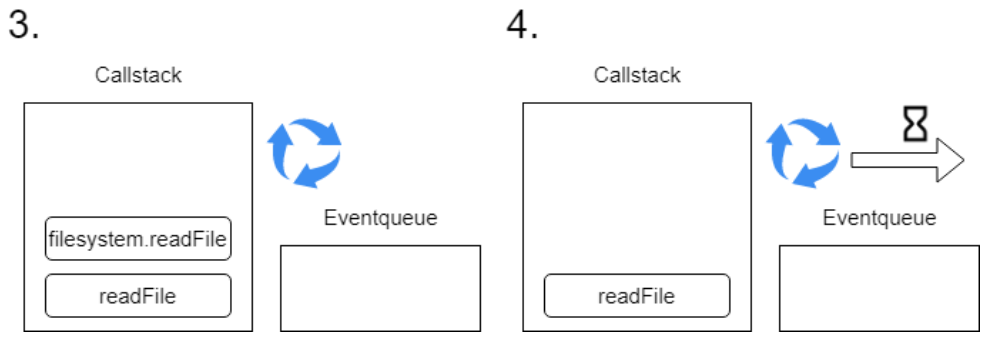
Die asynchrone Funktion heißt nun ebenfalls nur noch *readFile*. Sie nimmt jedoch noch einen zweiten Parameter entgegen. In diesem Fall eine Referenz auf eine Funktion, welche direkt inline und anonym definiert wird. Hierbei handelt es sich um eine Callbackfunktion. Sie wird aufgerufen nachdem die Datei vom Filesystem eingelesen wurde. Sollte dabei ein Fehler aufgetreten sein, wird dieser als erster Parameter übergeben. Ist kein Fehler aufgetreten hat *err* den Wert null und der Inhalt der Datei befindet sich als String in *data*. Im Rumpf der Callbackfunktion können diese Werte nun weiterverarbeitet werden.

Abbildung 6: Eventloop - Ablauf 1



Auch in diesem Beispiel ist der Callstack zu Beginn leer und es wird als erstes die Funktion *readFile* auf den Stack gelegt und ausgeführt.

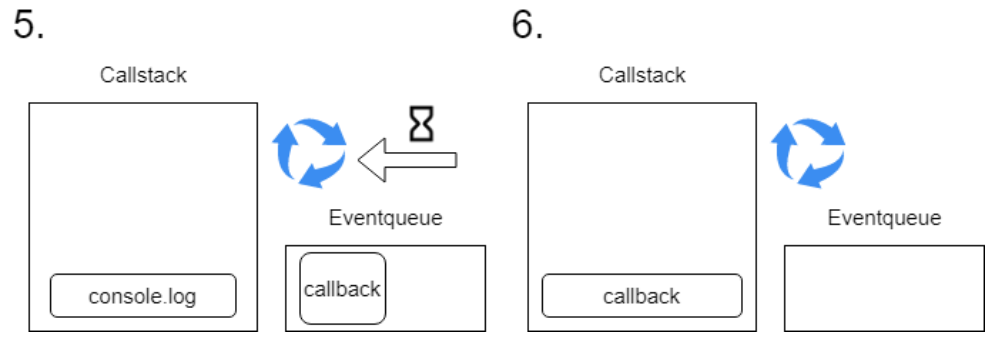
Abbildung 7: Eventloop - Ablauf 2



Da diese Funktion weitere Aufrufe enthält müssen diese ebenfalls dem Stack hinzugefügt werden. Dazu wird die *readFile*-Funktion des *filesystem*-Moduls im dritten Schritt auf den Stack gelegt. Sobald sie ausgeführt wird übergibt sie im vierten Schritt den Aufruf an die Schnittstelle zum Betriebssystem, welche durch libUV bereitgestellt wird und registriert dabei den Callback. Danach wird sie wieder vom Stack entfernt. Somit wurde das blockierende Einlesen der Datei an einen Thread des Threadpools von libUV übergeben, der Callstack ist somit nicht mehr blockiert und das Programm kann direkt fortgesetzt werden.[21]

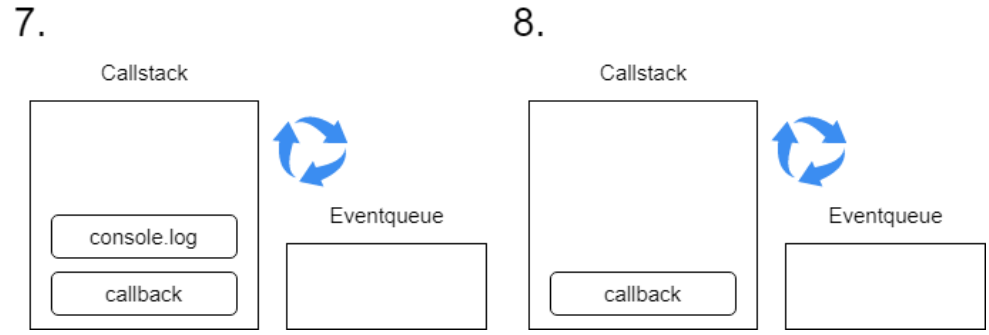
[21] [San19b].

Abbildung 8: Eventloop - Ablauf 3



Nun kann auch die eigens definierte *readFile*-Funktion vom Stack entfernt werden. Als nächstes folgt direkt der Aufruf der *console.log*-Funktion welche „Programm beendet“ auf die Konsole schreibt. Hierbei ist zu beachten, dass im Hintergrund das Betriebssystem immer noch damit beschäftigt ist, die Datei einzulesen. Jetzt wird auch die *console.log*-Funktion vom Stack entfernt. In der Zwischenzeit ist der Aufruf an das Betriebssystem zurückgekehrt und der Callback wird nun in der Eventqueue eingereiht. Ist der Callstack vollständig abgearbeitet, wird im achten Schritt die erste Funktion aus der Eventqueue entfernt und auf den Callstack gelegt.

Abbildung 9: Eventloop - Ablauf 4



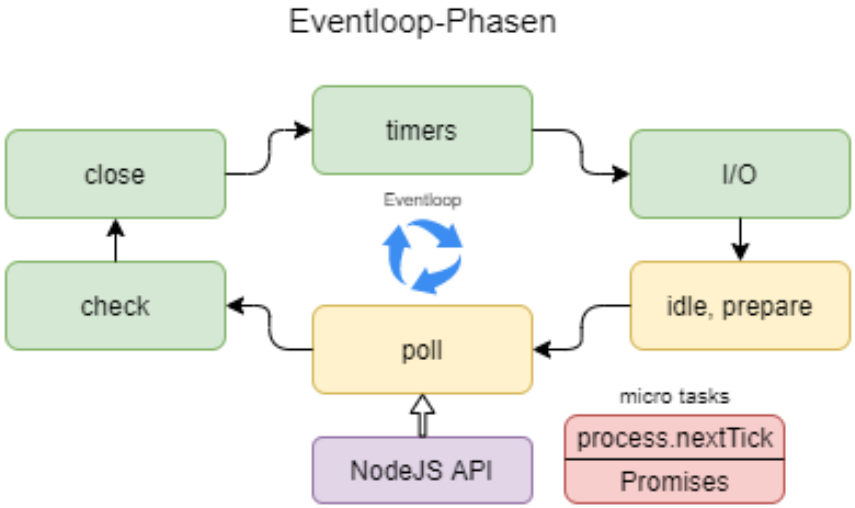
Nun befindet sich das Programm in Zeile 5 und *console.log* wird dem Stack hinzugefügt, ausgeführt und wieder entfernt. Außerdem kann die Callbackfunktion ebenfalls entfernt werden, da sie keine weiteren Aufrufe enthält. Nun in der Callstack und die Eventqueue leer und das Programm wird terminiert.

Man kann also zusammenfassend sagen, dass der Eventloop den Callstack überwacht. Nur wenn der Callstack leer ist, werden Callbackfunktionen in der

Eventqueue vom Eventloop, auf den Callstack gelegt.[22] In diesem Beispiel wurde das Konzept der Eventqueue stark vereinfacht. Hierbei handelt es sich nicht, wie im Beispiel gezeigt um eine einzelne Queue, sondern um 7 verschiedene. Jede dieser Queues arbeitet nach dem „first in, first out“ Prinzip und wird jeweils in verschiedenen Phasen des Eventloops abgearbeitet.[23] Für einen JavaScript-Entwickler ist wichtig dieses Konzept zu verstehen, da sonst schwerwiegende Fehler zu einem starken Performanceverlust führen können. LibUV stellt Ressourcen des Betriebssystems in JavaScript zur Verfügung auf die es selbst keinen Zugriff hat und somit nicht in JavaScript implementiert werden können. Zum Beispiel stellt libUV Schnittstellen für Dateisystemzugriffe, tcp/udp und Timer bereit, deren Callbacks jeweils in verschiedenen Phasen abgearbeitet werden.

Der Eventloop durchläuft folgende Phasen: timers, i/o callbacks, idle/prepare, poll, check, close callbacks und micro tasks.[24] Auf die “idle/prepare” Phase, wird im Rahmen dieser Arbeit nicht genauer eingegangen, da hier NodeJS interne Prozesse stattfinden. Hier hat der Programmierer, ebenso wie in der poll-Phase keinen direkten Einfluss, deshalb sind diese Phasen gelb markiert. Grün markierte Phasen lassen sich im Programm direkt beeinflussen.

Abbildung 10: Eventloop - Phasen



[22] Vgl. ebenda.
[23] [Log20].
[24] Vgl. ebenda.

In der timers-Phase werden alle Callbacks welche mit den Funktionen *setTimeout* und *setInterval* registriert wurden abgearbeitet.[25] Jede dieser Funktionen nimmt neben einer Referenz auf einen Callback auch eine Zeit in Millisekunden entgegen. Werden diese Funktionen aufgerufen, so wird der Aufruf direkt an libUV weitergegeben und dort in einem anderen Thread der Timer ausgeführt. Währenddessen werden in JavaScript die nächsten Zeilen des Programms synchron abgearbeitet. Diese Funktionen unterscheiden sich lediglich darin, dass der Callback bei *setTimeout* nur einmal nach der gegebenen Zeit ausgeführt wird und bei *setInterval* immer nach der gegebenen Zeit. Die *setInterval*-Funktion muss im Programmablauf explizit wieder gestoppt werden. Erst nach der übergebenen Zeit meldet die Schnittstelle dem Eventloop, dass der Callback nun in die timers-Queue eingereiht werden kann. Jetzt muss der Eventloop erst einmal die timers-Phase erreichen. Falls nicht schon andere Callbacks in der timers-Queue liegen und der Callstack leer ist, wird der Callback ausgeführt. Folgendes Beispiel soll verdeutlichen, dass es deshalb immer eine geringe Verzögerung gibt.[26]

Abbildung 11: Eventloop - *setTimeout*

```
1  const time = process.hrtime()
2  setTimeout(() => {
3      console.log(process.hrtime(time)[1]/1000000)
4  }, 0)
5
```

In der ersten Zeile wird die Zeitmessung gestartet. Hierbei handelt es sich um einen „high resolution timer“, welcher auf eine Nanosekunde genau eine Zeit messen kann. Als nächstes wird *setTimeout* aufgerufen. Hier wird wieder ein Callback übergeben und eine 0 als Dauer. Hier sollte man denken, dass der Callback unverzüglich ausgeführt wird. Wenn der Callback aufgerufen wurde wird die Zeit gestoppt und auf der Konsole ausgegeben.

[25] [Ope21c].
[26] [San19b].

Diese Zeit schwankt, abhängig von der genutzten Hardware extrem bei mehreren Versuchen, jedoch ist sie niemals 0. Diese Verzögerung wird als Eventlooplag bezeichnet.[27]

In der I/O-Phase werden alle Callbacks abgearbeitet, die z.B. aus Dateisystemzugriffen oder http-Anfragen resultieren.[28] Sie sind jedoch erst für den nächsten Durchlauf des Eventloops vorgesehen, da sie dort eingereiht wurden, als der Eventloop sich nicht in der poll-Phase befand.[29] Ist er in der poll-Phase angekommen werden alle I/O-Callbacks abgearbeitet bis diese Queue leer ist. Ist sie beim Eintreffen bereits leer, wird die check-Queue geprüft. Sollte diese ebenfalls leer sein, verweilt der Eventloop in der poll-Phase und wartet auf neue hereinkommende Callbacks von der NodeJS API und führt diese solange direkt aus bis entweder neue Callbacks in die check-Queue oder in die timers-Queue eingereiht werden. Wenn in dieser Zeit also ein Timer ausläuft und der Callback in die timers-Queue gelegt wird, überspringt der Eventloop die check-Phase, da die Queue ohnehin leer ist.[30]

Befindet sich der Eventloop die ganze Zeit in der poll-Phase und es gibt jedoch keine auslaufenden Timer, kann der Programmierer mittels *setImmediate* einen Callback in die check-Queue legen.[31] Das führt dazu, dass der Eventloop den aktuellen I/O-Callback abarbeitet und dann sofort in die check-Phase übergeht um dort die Queue abzuarbeiten. Hier kann der Programmierer also priorisieren. Es ist auch hier wieder zu beachten, dass jeder Callback wiederum Funktionen aufrufen kann und der Callstack somit wächst. Der Eventloop kann aber erst die nächste Callbackfunktion abarbeiten, wenn der Callstack leer ist. Das gilt für jeden Callback in jeder Phase. Hier wird deutlich das ein Phasenübergang des Eventloops an einige Bedingungen geknüpft ist. Zum einen muss die aktuelle Queue vollständig abgearbeitet sein. Das bedeutet das jeder Callback einzeln auf den Callstack gelegt und ausgeführt wird.[32]

---

[27] [Het19].
[28] [Log20].
[29] [Ope21c].
[30] Vgl. ebenda.
[31] Vgl. ebenda.
[32] [Gok21].

Die nächste Bedingung ist, dass der Callstack jedes Mal leer sein muss, damit der nächste Callback aus der Queue genommen werden kann. Ist die aktuelle Queue und der Callstack leer, so wird die microtask-Queue abgearbeitet.[33] Der Eventloop prüft dieses ebenfalls zwischen allen Phasen, da diese Queue ebenso leer sein muss, bevor mit der nächsten Phase fortgefahren werden kann. In die microtask-Queue werden alle Callbacks von *process.nextTick* oder von Promises eingereiht. [34] Promises sind ein Konstrukt aus dem ECMAScript-Standard von 2015 um asynchrones Programmieren in JavaScript übersichtlicher zu gestalten. Verhindert man als Programmierer, dass auch nur eine dieser Bedingungen erfüllt wird, bleibt der Eventloop in der aktuellen Phase hängen und ist blockiert. Das kann z.B. geschehen indem Funktionen immer wieder neue Funktionen aufrufen und der Callstack somit sehr stark anwächst. Außerdem ist es möglich, dass das Programm sehr lange in einer Funktion verweilt und somit das abarbeiten des Callstacks verhindert. Dies geschieht etwa durch sehr komplexe Berechnungen. Auch das immer wieder erneute Einreihen von Callbackfunktionen in die microtask-Queue sorgt dafür, dass der Eventloop nicht fortgesetzt wird, da diese immer unverzüglich verarbeitet werden. Dieses Szenario kommt etwa einer Endlosschleife gleich und wird „I/O starvation“ genannt. [35] Es kommt zum „Aushungern“ der I/O Verarbeitung.

[33] Vgl. ebenda.
[34] Vgl. ebenda.
[35] Vgl. ebenda.

3 NodeJS Optimierungen

Im Folgenden Kapitel werden einige Optimierungen von NodeJS-Anwendungen behandelt und getestet. Alle Tests wurden unter der gleichen Testumgebung durchgeführt. Dabei handelt es sich um ein System mit einem Intel Core i7-8565U Prozessor und 8 Gigabyte DDR4 Arbeitsspeicher. Weiterhin wird bei jedem Test stets NodeJS in der Version 14.10.1 verwendet. Weiterhin finden einige Tests innerhalb eines Webservers statt, da die Performance anhand der bearbeiteten Anfragen pro Sekunde gut messbar ist und es viele Testtools gibt, welche einen Webserver auf Basis dieser Metrik testen können.

3.1 Korrekte Konfiguration der NodeJS-Umgebung

Am einfachsten und schnellsten kann man Performance durch das korrekte Setzen von Umgebungsvariablen gewinnen. Dabei ist „NODE_ENV“ die wichtigste Umgebungsvariable in der NodeJS-Welt. V8 und libUV selbst greifen beim Start einer NodeJS-Anwendung darauf zu und aktivieren einige Optimierungen. Auch viele Bibliotheken des Node Package Managers greifen zur Laufzeit darauf zu und aktivieren ihrerseits einige Optimierungen wie z.B. Caching. Sie sollte im Produktionsbetrieb auf „production“ und während der Entwicklung der Anwendung auf „development“ gesetzt werden. Wird sie vom Entwickler gar nicht gesetzt steht sie per Standardkonfiguration auf „development“. Das kann im Produktionsbetrieb zu einem massiven Performanceverlust führen.

3.1.1 **Testaufbau**

Der Testaufbau beinhaltet im Wesentlichen zwei NodeJS-Anwendungen. Eine davon stellt einen einfach Webserver mittels Express auf dem lokalen System bereit.

Abbildung 12: Test mit Umgebungsvariablen - Express

```
const express = require("express")
const app = express();


app.get('/', (req, res) => {
    let a = 1;
    for (let i = 0; i < 100; i++){
        a+=i*a
    }
    res.json({a})
})

app.listen(8080)
```

Der Webserver läuft auf dem Port „8080“ und enthält einen Endpunkt welcher über die URL „http://localhost:8080/“ erreichbar ist. Sollte dieser Endpunkt aufgerufen werden wird in einer Schleife eine Berechnung durchgeführt und das Ergebnis zurückgegeben. Die zweite Anwendung enthält eine Bibliothek namens „autocannon“. Diese ist in der Lage parallele Anfragen an eine Zieladresse auszuführen und das Ergebnis auf der Konsole auszugeben. Da autocannon vielseitig konfigurierbar ist, wurden die Einstellungen in ein JavaScript eingepflegt um sicherzustellen, dass jeder Test unter identischen Bedingungen ausgeführt wird.

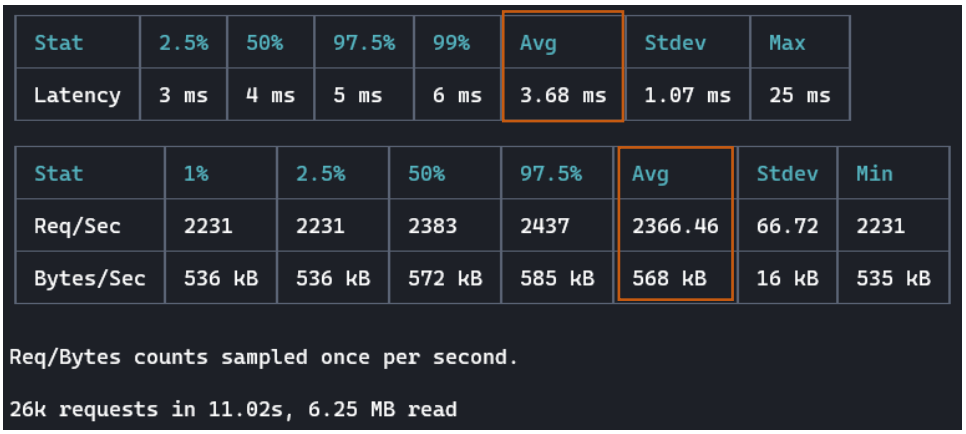
Abbildung 13: Test mit Umgebungsvariablen - Autocannon

```
const autocannon = require("autocannon")
autocannon({
    url: 'http://localhost:8080',
    connections: 10,
    pipelining: 1,
    duration: 10
  }, (err, result) => {
    console.log(autocannon.printResult(result))
  })
```

In dieser Konfiguration werden über 10 Verbindungen während 10 Sekunden parallel möglichst viele Anfragen an den lokalen Webserver gesendet. Dabei wird die Umgebungsvariable „NODE_ENV“ einmal auf „production“ und einmal auf „development“ gesetzt.

3.1.2 Testauswertung

Abbildung 14: Test mit Umgebungsvariablen - Auswertung 1



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	3 ms	4 ms	5 ms	6 ms	3.68 ms	1.07 ms	25 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	2231	2231	2383	2437	2366.46	66.72	2231
Bytes/Sec	536 kB	536 kB	572 kB	585 kB	568 kB	16 kB	535 kB

Req/Bytes counts sampled once per second.

26k requests in 11.02s, 6.25 MB read

Der erste Test mit „NODE_ENV=development“ zeigt, dass im Durchschnitt ca. 2366 req/sec (Anfragen pro Sekunde) beantwortet wurden. Dabei betrug die durchschnittliche Latenz etwa 3,5 ms.

Abbildung 15: Test mit Umgebungsvariablen - Auswertung 2

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	0 ms	0 ms	1 ms	1 ms	0.09 ms	0.34 ms	14 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	7891	7891	14471	14943	13775.8	2009.24	7890
Bytes/Sec	1.89 MB	1.89 MB	3.47 MB	3.59 MB	3.31 MB	482 kB	1.89 MB

Req/Bytes counts sampled once per second.

138k requests in 10.02s, 33.1 MB read

Setzt man nun „NODE_ENV“ auf „production“ werden ca. 13776 req/sec beantwortet und das bei einer durchschnittlichen Latenz von unter einer Millisekunde. Hier konnten also fast 6-mal mehr Anfragen beantwortet werden. Außerdem interessant ist, dass bei diesem Test insgesamt 138 tausend Anfragen in 10 Sekunden erfolgreich beantwortet wurden. Bei dem ersten Test waren es nur 26 tausend in 11 Sekunden.
Die Tests zeigen also eindeutig, dass „NODE_ENV“ in Produktionsumgebungen explizit auf „production“ gesetzt werden sollte.

3.2 Compiler Optimierungen

Wie in Kapitel 2.1 bereits erwähnt wurde, besteht V8 aus zwei verschiedenen Compilern bzw. Interpretern. Letzterer sorgt für eine starke Optimierung des Codes auf Basis von Annahmen, wie beispielweise Funktionen zukünftig genutzt werden könnten. Sollten diese Annahmen im späteren Programmverlauf nicht mehr zutreffen, muss der optimierte Code verworfen und der normale Code genutzt werden. Dies gelingt dadurch, dass der Compiler zu jedem Objekt im Hintergrund sogenannte „verstecke Klassen“ erstellt.[36] Ändert sich der Typ eines Objektes, da ein neues Attribut zur Laufzeit hinzugefügt wurde, muss eine neue versteckte Klasse erstellt werden. [37] Außerdem gibt es auch Codeabschnitte bei denen es dem Compiler nicht möglich ist Optimierungen vorzunehmen. Solche Konstrukte sollten also weitestgehend vermieden werden, wenn dadurch die Komplexität des Codes nicht wesentlich zunimmt. Das Folgenden Beispiel soll zeigen, wie Optimierungen in V8 funktionieren.

Abbildung 16: Compileroptimierung - Codebeispiel

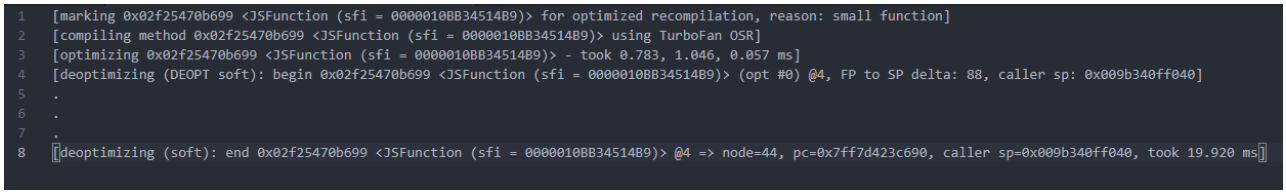
```
1  function add(a,b) {
2      return a + b;
3  }
4
5  for (let i = 0; i < 1000000; i++) {
6      add(i,i);
7  }
8
9  add("", "")
```

Es wird eine einfache Funktion definiert welche anschließend in einer Schleife häufig aufgerufen wird und dabei Zahlen als Parameter übergeben bekommt. Abschließend wird sie einmal außerhalb der Schleife mit einem jeweils einem leeren String als Parameter aufgerufen.

[36] [Wan19].
[37] Vgl. ebenda.

Startet man nun das Programm mit folgendem Befehl „node --trace-deopt --trace-opt index.js“ werden zusätzlich alle Optimierungen/Deoptimierungen vom Compiler auf der Konsole ausgegeben. [38] Das führt zu folgendem Ausschnitt der Konsolenausgabe.

Abbildung 17: Compileroptimierung - Konsolenausgabe



Es ist gut zusehen, dass der Compiler die Funktion zur Optimierung neu kompiliert und nutzt um alle Aufrufe innerhalb der Schleife zu bearbeiten. Trifft er nun auf den letzten Aufruf, passt die Optimierung nicht mehr und kann daher nicht mehr genutzt werden. Die Optimierung muss also Rückgängig gemacht werden. In der letzten Zeile ist sogar am Ende sichtbar, dass dieser Vorgang ca. 20ms gedauert hat. Stellt man sich nun vor, dass das Programm im Kontext eines Webservers mehrere hundertmal an dieser Stelle vorbeikommt und die Funktion jedes Mal optimiert wird und die Optimierung danach verworfen werden muss, kann hier viel Zeit gespart werden, indem darauf geachtet wird, Funktionen immer mit den gleichen Typen auszurufen. Es existieren noch weitere ähnlich geartete Beispiele auf die es in diesem Kontext zu achten gilt. Zum Beispiel ist dem Compiler nicht möglich Funktionen zu optimieren, die ein try-catch-Block enthalten, da hier angenommen wird, dass jeder Zeit ein Fehler passieren kann, trifft der Compiler keine Annahmen auf deren Basis optimiert werden könnte.[39]

[38] [Tur17].
[39] Vgl. ebenda.

3.3 Eventloop Blockaden vermeiden

Einer der wichtigsten Punkte bei der Entwicklung von NodeJS-Anwendungen ist es, zu vermeiden, dass der Eventloop blockiert ist. Bei einem Webserver kann dies dazu führen, dass keine anderen Anfragen beantwortet werden. Wie bereits in Kapitel 2.2 erwähnt, besteht der Eventloop aus mehreren Phasen. Dabei kann er immer nur zur nächsten Phase übergehen, wenn der Callstack und die aktuelle Eventqueue vollständig leer sind. Das bedeutet das rechenintensive Funktionen für ein langsameren Zyklus des Eventloops sorgen. Folgender Test soll eine rechenintensive Funktion in einem Webserver mit der Express-Bibliothek simulieren.

3.3.1 Testaufbau

Ähnlich dem Testaufbau aus Kapitel 3.1.1 kommt wieder ein Expressserver zum Einsatz. Beim Aufruf soll eine Schleife mit einer großen Anzahl an Durchläufen eine rechenintensive Aufgabe simulieren. Je mehr Durchläufe desto komplexer die Aufgabe.

Abbildung 18: Eventloop Block - Codebeispiel

```
1  app.get('/', (req, res) => {
2      for(let i= 0; i<LARGE_NUBMER; i++) {
3          // heavy work here
4      }
5      res.send("Done")
6  })
7
8
9  app.listen(8080)
```

Im ersten Test wird die Schleife direkt synchron abgearbeitet. Die Callbackfunktion liegt also sehr lange auf dem Callstack. Dadurch kann der Eventloop nur langsam zirkulieren.

Abbildung 19: Eventloop Block - Worker

```
app.get('/', async (req, res) => {
    const result = await createWorker(LARGE_NUBMER)
    res.send(result)
})

const createWorker = (workerData) => {
    return new Promise((resolve, reject) => {
        const worker = new Worker(path.resolve(__dirname, 'worker.js'), {workerData});
        worker.on('message', resolve);
    });
}

```

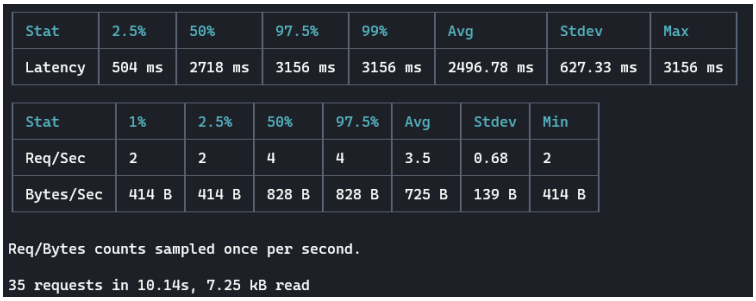
Im zweiten Test übernimmt ein sogenannter „worker“ die Aufgabe, die Schleife zu bearbeiten.[40] Er bekommt lediglich die Anzahl der Durchläufe übergeben und liefert später das Ergebnis zurück. Das Ganze geschieht asynchron in einem der von libUV bereitgestellten Threads. Somit ist der Webserver in der Lage weitere Anfragen zu bearbeiten.

[40] [Köl19].

3.3.2 **Testauswertung**

Mit dem Tool „autocannon“ wird versucht über 10 parallele Verbindungen hinweg für 10 Sekunden so viele Anfragen wie möglich zu stellen und antworten zu erhalten.

Abbildung 20: Eventloop Block - Auswertung 1



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	504 ms	2718 ms	3156 ms	3156 ms	2496.78 ms	627.33 ms	3156 ms

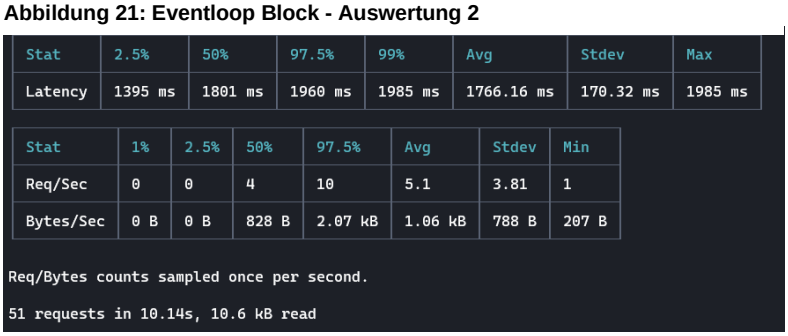
Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	2	2	4	4	3.5	0.68	2
Bytes/Sec	414 B	414 B	828 B	828 B	725 B	139 B	414 B

Req/Bytes counts sampled once per second.

35 requests in 10.14s, 7.25 kB read

Im ersten Versuch werden nur 35 Anfragen bei einer durchschnittlichen Latenz von ca. 2,5s abgearbeitet.

Abbildung 21: Eventloop Block - Auswertung 2



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1395 ms	1801 ms	1960 ms	1985 ms	1766.16 ms	170.32 ms	1985 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	0	0	4	10	5.1	3.81	1
Bytes/Sec	0 B	0 B	828 B	2.07 kB	1.06 kB	788 B	207 B

Req/Bytes counts sampled once per second.

51 requests in 10.14s, 10.6 kB read

Im zweiten Versuch schafft der Eventloop mehr Zyklen und kann daher 51 Anfragen mit einer etwas geringeren Latenz von ca. 1,7 Sekunden bearbeiten. Setzt man die Zahl der Schleifeniterationen herunter, so ist Variante ohne „worker“ etwas schneller. Das legt nahe, dass das erstellen eines „worker-threads“ etwas Overhead erzeugt was den Prozess etwas verlangsamt. Bei kleinen Funktionen fällt dieser Overhead etwas mehr ins Gewicht, so dass „worker“ erst bei sehr komplexen Berechnungen sinnvoll erscheinen. Genauer untersucht werden muss hier die Möglichkeit, das Erstellen eines „workers“ effizienter zu gestalten. So könnte ein vorher erstellen Pool aus „worker-threads“ die Aufgaben übernehmen und es müsste nicht bei jeder Anfrage ein neuer „worker“ erstellt werden. So könnte eventuell noch etwas Zeit gespart werden.

3.4 Express vs. Fastify

Sollte in NodeJS ein Webserver benötigt werden, kommt in den meisten Fällen die Bibliothek Express zum Einsatz. Das sieht man beispielsweise an den über 16 Millionen wöchentlichen Downloads (Stand: 16.3.2021) des Pakets aus der offiziellen npm-Bibliothek. Es gibt jedoch verschiedene Alternativen zu Express, welche auf unterschiedliche Art und Weise implementiert wurden. Eine Alternative davon ist eine Bibliothek namens „fastify“. Sie wurde daraufhin optimiert Anfragen möglichst schnell zu bearbeiten. Dazu wurde das Beispiel aus Kapitel 3.1.1 angepasst und „express“ durch „fastify“ ersetzt. Mit der gleichen Testumgebung wie in Kapitel 3.1.2 wurden über 10 Verbindungen für 10 Sekunden möglichst viele Anfragen gestellt.

Abbildung 22: Fastify - Auswertung

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	0 ms	0 ms	0 ms	1 ms	0.03 ms	0.3 ms	31 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	12199	12199	18975	20127	18612.41	2236.46	12199
Bytes/Sec	2.14 MB	2.14 MB	3.32 MB	3.52 MB	3.26 MB	391 kB	2.13 MB

Req/Bytes counts sampled once per second.

186k requests in 10.05s, 32.6 MB read

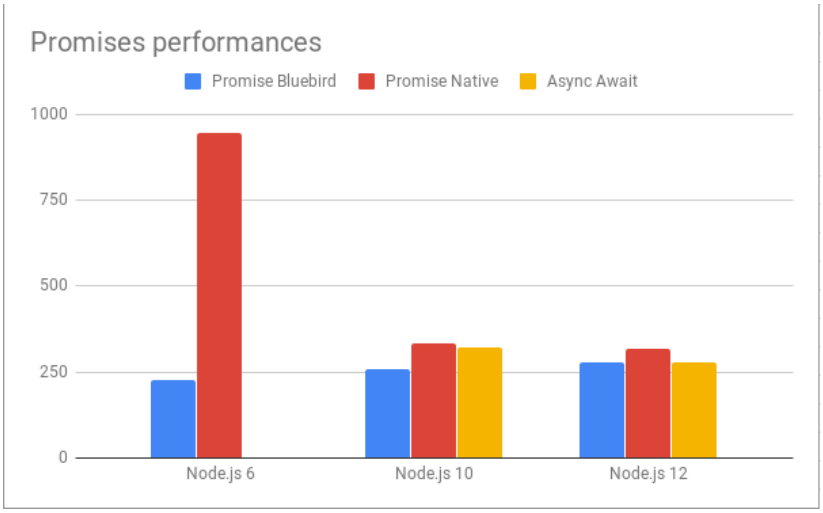
Mit fastify konnten fast 19000 req/sec beantwortet werden. Mit insgesamt 186 tausend Anfragen sind es 48 tausend mehr als mit Express. Das Ganze stellt jedoch kein reales Szenario aus der Produktivumgebung dar. Wie sich der Datendurchsatz dort verhält kann nur in solch einer Umgebung Live getestet werden.

3.5 Weitere Möglichkeiten zur Optimierung

In diesem Kapitel sollen weitere Möglichkeiten aufgezeigt werden, um effizientere NodeJS-Anwendungen zu entwickeln. Diese Möglichkeiten wurden beim Testen und Analysieren vorangegangener Beispiele ebenfalls gefunden aber im Rahmen dieser Arbeit, aufgrund von etwas höherer Komplexität nicht weiter analysiert. Trotzdem sollten diese Möglichkeiten auch in Erwägung ziehen, sollte es einmal zu Performanceproblemen kommen.

Die einfachste dieser Möglichkeiten ist, immer die aktuellste Version von NodeJS zu nutzen. Viele Konstrukte wie z.B. Schleifen, Manipulation von Arrays, Objektzugriffe oder das Auflösen von Promises werden mit jeder Version effizienter. Folgende Abbildung zeigt einen Vergleich der Latenzen von Promises unter verschiedenen NodeJS Versionen. Es ist deutlich zu erkennen, dass die Latenzzeiten mit neuen Versionen immer weiter sinken.

Abbildung 23: Promiseperformance verschiedener NodeJS Versionen



Quelle: [Mar19]

Eine weitere und noch nicht sehr verbreitete Möglichkeit liegt in der Entwicklung von Modulen für NodeJS in der Sprache C++. Eine neue API von NodeJS ermöglicht es C++-Code in NodeJS als Modul einzubinden und darin sehr komplexe Berechnungen in einem anderen Thread asynchron durchzuführen.[41] Das Ganze ähnelt den „worker-threads“ aus Kapitel 3.3 und wird ebenfalls in den Threadpool von libUV ausgelagert. Hier besteht ebenfalls eine Möglichkeit NodeJS an das System anzupassen und somit effizienter zu nutzen. Mit der Umgebungsvariable „UV_THREADPOOL_SIZE“ kann die Größe des Threadpools von libUV angepasst werden.[42] Dieser wird in der Standardkonfiguration nur mit vier Threads initialisiert. Die meisten Systeme bieten jedoch mehr als vier Threads. Die Anzahl kann also abhängig vom System erhöht werden um alle Ressourcen effektiv zu nutzen. Dabei sollte jedoch ein Thread für das System selbst übrigbleiben.

Weiterhin sollte man in kleineren Umgebungen mit wenig Arbeitsspeicher in Erwägung ziehen die Garbage Collection etwas zu optimieren, indem man zum einen alle Objekte im Programm immer im kleinstmöglichen Scope definiert. Sollte ein Objekt nicht mehr benötigt werden kann es vom Garbage Collector entfernt werden. Globalere Objekte werden meistens länger im Speicher gehalten, da darauf immer noch eine Referenz verweist. Zum anderen kann mit dem Startparameter „max-old-space-size“ die Größe des verfügbaren Speichers in V8 festlegen.[43] Sollte eine NodeJS-Anwendung an dieses Limit kommen, versucht der Garbage Collector ungenutzten Speicher frei zu räumen. Wählt man diesen Wert zu klein, verbraucht der Garbage Collector häufiger Ressourcen um seine Arbeit zu verrichten. Wählt man diesen Wert zu groß kann es zum Absturz der Anwendung führen, wenn im Gesamtsystem nicht genügend Speicher vorhanden ist. In einer Microserviceumgebung in der viele NodeJS-Anwendungen in kleinen Containern mit wenig Speicher ausgeführt wird, sollte eine Anpassung des Parameters in Betracht gezogen werden.

---

[41] [Ber18].
[42] [Ope21a].
[43] Vgl. ebenda.

4 Nutzbarkeit solcher Techniken in Produktivsystemen

Es gibt unzählige Möglichkeiten NodeJS-Anwendungen an verschiedenen Stellen zu optimieren und vergleichsweise wenig Aspekte auf die man bei der Entwicklung solcher Anwendungen tatsächlich noch achten muss, um guten Code zu produzieren. NodeJS nimmt dem Entwickler mittels einiger Technologien in V8 und libUV viel Optimierungsaufwand ab und ist dadurch für I/O intensive Aufgabe sehr gut geeignet. Als Entwickler muss lediglich stets darauf geachtet werden, dass der Eventloop nicht blockiert wird. Nach genauerer Analyse der NodeJS-Architektur stellt sich das als wichtigstes Merkmal heraus. Darüber hinaus sollte man einige Techniken kennen, sehr rechenintensive synchrone Prozesse in einen anderen Thread oder am besten noch auf eine ganze andere Ressource auszulagern und dorthin nur mittels http zu kommunizieren um Daten zur Verarbeitung zu senden und auf das Ergebnis zu warten. Dort liegen schließlich die Stärken von NodeJS.

In größeren Produktivsystemen finden solche Techniken nur im geringen Maße Anwendung. NodeJS wird meistens in Microservice-Architekturen als kleiner Webserver betrieben. Diese Webserver kommunizieren dann untereinander um verschiedene Prozesse komplett abzubilden, übernehmen dabei aber nur ein Teilproblem des großen Ganzen. Darum haben sie oft nur eine kleine und einfache Aufgabe, sodass der dazugehörige Programmcode sehr simpel und einfach zu warten ist. Die Optimierungsmöglichkeiten sind bei kleinerem Code verschwindend gering, da wie bereits erwähnt NodeJS schon einen großen Teil zur Optimierung beiträgt. Sollte es in diesem Umfeld jedoch an einer speziellen Stelle gefordert sein, dass eine NodeJS-Anwendung in äußert kurzer Zeit ein Ergebnis liefert, gibt es viele Möglichkeiten den Code mit etwas Aufwand zu optimieren. Ein mögliches Beispiel könnten viele parallele Echtzeitberechnungen von Preisen in einem Onlineshop sein. Hierbei ist aber zu beachten, dass solche Optimierungen meistens mit einem viel komplexeren und schlechter wartbaren Programmcode einher gehen. Somit sollte bei der Entwicklung in großen Teams immer der Nutzen einer komplexen Optimierung im Vergleich zur Wartbarkeit abgewogen werden.

III Literaturverzeichnis

[Aug20] Augsten, Stephan: „Was ist Node.js?“, 2020.
https://www.dev-insider.de/was-ist-nodejs-a-972703/
Abruf: 2021.03.24

[Ber18] Berger, Stefan: „NodeJS: Wie bekomme ich nativen Code in NodeJS zum Laufen?“, 2018.
https://codefluegel.com/blog/run-native-code-in-nodejs/
Abruf: 2021.03.24

[Gok21] Gokhman, Michael: „Node.js Event-Loop: How even quick Node.js async functions can block the Event-Loop, starve I/O | Snyk Engineering“.
https://snyk.io/blog/nodejs-how-even-quick-async-functions-can-block-the-event-loop-starve-io/
Abruf: 2021.03.24

[Goo21] Google: „Projects – opensource.google“.
https://opensource.google/projects/v8
Abruf: 2021.03.24

[Hax20] Haxhi, Gerald: „Node.js Internals: Libuv and the event loop behind the curtain“, 2020.
https://medium.com/softup-technologies/node-js-internals-libuv-and-the-event-loop-behind-the-curtain-30708c5ca83
Abruf: 2021.03.24

[Het19] Hettler, David: „Monitoring Node.js: Watch Your Event Loop Lag!“, 2019.
https://davidhettler.net/blog/event-loop-lag/
Abruf: 2021.03.24

[Kan20] Kaneriya, Tejas: „What is Node.js? Where, When & How To Use It (With Examples)“, 2020.
https://www.simform.com/what-is-node-js/
Abruf: 2021.03.24

[Köl19] Kölpin, Sven: „Node.js: Unblock the Event-Loop | Informatik Aktuell“, 2019.
https://www.informatik-aktuell.de/entwicklung/programmiersprachen/nodejs-unblock-the-event-loop.html
Abruf: 2021.03.24

[Log20] LogRocket Blog: „A deep dive into queues in Node.js - LogRocket Blog“, 2020.
https://blog.logrocket.com/a-deep-dive-into-queues-in-node-js/
Abruf: 2021.03.24

[Mar19] Maret, Adrien: „Bluebird vs Native vs Async/Await - State of Promises performances in 2019“, 2019.
https://blog.kuzzle.io/bluebird-vs-native-vs-async/await-state-of-promises-performances-in-2019
Abruf: 2021.03.24

[Mir21] Mirza, Mohammad Shad: „Nodejs Lesson 15: Internals of Nodejs: LibUV“, 2021.
https://soshace.com/16-node-js-lessons-event-loop-libuv-library-pt-1/
Abruf: 2021.03.24

[Moz21] Mozilla: „Dynamic typing - MDN Web Docs Glossary: Definitions of Web-related terms | MDN“.
https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing
Abruf: 2021.03.24

[Ope20] OpenJS Foundation: „The V8 JavaScript Engine“, 2020.
https://nodejs.dev/learn/the-v8-javascript-engine
Abruf: 2021.03.24

[Ope21] OpenJS Foundation: „Command-line options | Node.js v15.12.0 Documentation“, 2021.
https://nodejs.org/api/cli.html#cli_uv_threadpool_size_size
Abruf: 2021.03.24

[Ope21] OpenJS Foundation: „Dependencies | Node.js“, 2021.
https://nodejs.org/en/docs/meta/topics/dependencies/#libuv
Abruf: 2021.03.24

[Ope21] OpenJS Foundation: „The Node.js Event Loop, Timers, and process.nextTick() | Node.js“, 2021.
https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
Abruf: 2021.03.24

[Rad20] Radziwill, Marc: „Mastering JavaScript high performance in V8“, 2020.
https://marcradziwill.com/blog/mastering-javascript-high-performance/
Abruf: 2021.03.24

[San19] Santos, Lucas: „Node.js Under The Hood #2 - Understanding JavaScript“, 2019.
https://dev.to/khaosdoctor/node-js-under-the-hood-2-understanding-javascript-48cn
Abruf: 2021.03.24

[San19] Santos, Lucas: „Node.js Under The Hood #3 - Deep Dive Into the Event Loop“, 2019.
https://dev.to/khaosdoctor/node-js-under-the-hood-3-deep-dive-into-the-event-loop-135d
Abruf: 2021.03.24

[San19] Santos, Lucas: „Node.js Under The Hood #4 - Let's Talk About V8“, 2019.
https://dev.to/khaosdoctor/node-js-under-the-hood-4-let-s-talk-about-v8-1eol
Abruf: 2021.03.24

[San19] Santos, Lucas: „Node.js Under The Hood #7 - The new V8“, 2019.
https://dev.to/khaosdoctor/node-js-under-the-hood-7-the-new-v8-4gd6
Abruf: 2021.03.24

[San20] Santos, Lucas: „Node.js Under The Hood #10 - Compiler Optimizations!“, 2020.
https://dev.to/khaosdoctor/node-js-under-the-hood-10-compiler-optimizations-5dol
Abruf: 2021.03.24

[Sid20] Siddique, Sharjeel: „What does it mean by Javascript is single threaded language“, 2020.
https://medium.com/swlh/what-does-it-mean-by-javascript-is-single-threaded-language-f4130645d8a9
Abruf: 2021.03.24

[Tur17] Turckheim, Vladimir de: „How to find Node.js Performance Optimization Killers | @RisingStack“, 2017.
https://community.risingstack.com/how-to-find-node-js-performance-optimization-killers/
Abruf: 2021.03.24

[Wan19] Wan, Alvin: „How JavaScript works: Optimizing the V8 compiler for efficiency - LogRocket Blog“, 2019.
https://blog.logrocket.com/how-javascript-works-optimizing-the-v8-compiler-for-efficiency/
Abruf: 2021.03.24

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Studienarbeit mit dem Thema:

   ohne fremde Hilfe angefertigt 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 und
3. dass ich meine Studienarbeit bei keiner anderen Prüfung vorgelegt habe.

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

_____ _____

Ort, Datum