UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA - CCET
DEPARTAMENTO DE COMPUTAÇÃO - DC

Julio Cesar dos Santos Oliveira Filho

Um estudo sobre o uso de módulos core em testes automatizados de aplicações Node.js

SÃO CARLOS - SP
2024

Julio Cesar dos Santos Oliveira Filho

Um estudo sobre o uso de módulos core em testes automatizados de aplicações Node.js

Trabalho de conclusão de curso apresentada ao Departamento de Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em Ciência da Computação.

Orientador: Prof. Dr. André Takeshi Endo

SÃO CARLOS - SP
2024

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Cienêcias Exatas e de Tecnologia - CCET
Departamento de Computação

Comissão avaliadora

Membros da comissão examinadora que avaliou e aprovou a Defesa do Trabalho de Conclusão de Curso do candidato Julio Cesar dos Santos Oliveira Filho, realizada em 01/02/2024

Prof. Dr. André Takeshi Endo
Instituição: Universidade Federal de São Carlos

Prof. Dr. Daniel Lucrédio
Instituição: Universidade Federal de São Carlos

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Instituição: Universidade Federal de São Carlos

Dedicatória

Dedico este trabalho aos meus pais Sonia e Julio, minha avó Ana e todos meus familiares que me apoiaram nessa difícil e longa caminhada.

AGRADECIMENTO

Agradeço primeiramente aos meus pais Sonia e Julio e minha avó Ana, que foram os pilares da minha vida até eu conseguir encontrar e seguir o meu próprio caminho.

Agradeço meu tio Juninho, minha tia Lígia e minha prima Jhenifer, que sempre me motivaram a seguir focado nos objetivos.

Agradeço as minhas irmãs Juliana e Mariana e minhas tias Ana e Rosimeire, que sempre me apoiaram e auxiliaram em realizar esse objetivo da graduação em Ciência da Computação.

Agradeço ao Prof. Dr. André Takeshi Endo, por me oferecer todo suporte e me guiar nesse trabalho.

Agradeço ao Departamento de Computação, por todo os ensinamentos passados e papel na minha caminhada durante esta graduação.

Também agradeço por todos que fizeram parte da minha caminhada durante a realização do curso.

Você é você. Ser como você não é tão ambíguo quanto isso. Não importa o que você faça, não importa como você mude, isso não significa nada. Você é apenas você, não importa o quê. - Kaori Miyazono

RESUMO

Com o avanço contínuo das aplicações web, compreender seus comportamentos e operações chaves é indispensável para o aprimoramento das tecnologias utilizadas. Este estudo concentra-se na análise das funções assíncronas em casos de testes de um conjunto de aplicações reais utilizando Node.js. Para isto, foi estudado especificamente os módulos core do Node.js, explorando suas funções e padrões assíncronos.

Inicialmente, foram selecionados os projetos Node.js para análise com base em critérios definidos. Logo após, os testes dos projetos foram executados por um protótipo, sendo documentados em registros JSON suas funções assíncronas. Por fim, foi construída uma ferramenta de leitura destes registros, cujo objetivo era analisar e gerar novos dados para este trabalho, sendo transcritos em forma de planilhas e gráficos.

A partir de uma extensa quantidade de dados gerados, foram respondidas quatro questões de pesquisa relacionadas à: utilização dos módulos core, proporção de cada módulo core em relação ao total de funções assíncronas, padrões assíncronos mais frequentes e as funções mais usadas por módulos.

Por fim, foi observado um certo padrão de uso das funções assíncronas, sendo alguns módulos preteridos em relação a outros. Além disso, foi identificada uma disparidade significativa entre os dois padrões assíncronos mais utilizados, com o primeiro apresentando um uso consideravelmente maior que o segundo.

Palavras-chave: Funções assíncronas. Testes automatizados. JavaScript. Node.js. Módulos core.

ABSTRACT

With the continuous advancement of web applications, understanding their behaviors and key operations is indispensable for improving the technologies used. This study focuses on the analysis of asynchronous functions in test cases of a set of real applications using Node.js. Specifically, the core modules of Node.js were studied, exploring their functions and asynchronous patterns.

Initially, Node.js projects were selected for analysis based on defined criteria. Subsequently, the projects' tests were executed by a prototype, and their asynchronous functions were documented in JSON records. Finally, a tool for reading these records was built, aiming to analyze and generate new data for this work, transcribed in the form of spreadsheets and charts.

From a large amount of generated data, four research questions related to the use of core modules, the proportion of each core module in relation to the total asynchronous functions, the most frequent asynchronous patterns, and the most used functions by modules were answered.

In conclusion, a certain pattern of use of asynchronous functions was observed, with some modules being preferred over others. Moreover, a significant disparity between the two most used asynchronous patterns was identified, with the first showing considerably greater use than the second.

Keyword: Asynchronous functions. Automated testing. JavaScript. Node.js. Core modules.

Lista de Figuras

Lista de Tabelas

Conteúdo

1 INTRODUÇÃO

Há alguns anos, o JavaScript vem sendo a linguagem mais popular na plataforma GitHub; isto demonstra a popularidade desta linguagem de programação (Daigle, 2023). Ainda nessa lista, o TypeScript, que adiciona tipagem ao JavaScript, está em terceiro lugar, superando o Java no ano de 2023. Esta linguagem é muito utilizada na criação de interfaces gráficas para o lado do cliente. Para lidar com o lado do servidor, foi desenvolvida a plataforma Node.js.

O Node.js é uma plataforma altamente escalável e rápida para se construir o lado servidor de uma aplicação, sendo baseado na *engine* V8, que é atualmente desenvolvida pelo Google e utilizado no navegador Google Chrome. Esta plataforma tem se tornado popular para criação de aplicações *back-end*, possuindo o principal destaque na forma de execução *single-thread* (Node.js, 2023). Portanto, suas funções assíncronas não precisam bloquear a *thread* principal, permitindo ao desenvolvedor um maior controle. O Node.js oferece com ele um conjunto de módulos essenciais para seu funcionamento, conhecido como módulos core (Node.js, 2023). Nestes, é disponibilizada uma série de funções assíncronas essenciais para o desenvolvimento de aplicações reais.

Entender as funções assíncronas em aplicações de software é crucial para a sua manutenção e escalabilidade. Uma metodologia eficaz para alcançar essa compreensão é com a criação de uma grande quantidade de cenários, para entender como as funções assíncronas respondem. Para alcançar esses cenários, os testes automatizados são ideais. Esses testes simulam diversos casos, permitindo aos desenvolvedores observar como a aplicação se comporta sob diferentes condições. Portanto, com o avanço do desenvolvimento de um sistema, os testes automatizados oferecem uma forma de aplicar cenários complexos para entender o seu funcionamento, pois o seu intuito é de validação dos resultados obtidos.

Portanto, entender como o Node.js lida com o comportamento assíncrono é essencial para o aprimoramento de ferramentas e o entendimento de suas aplicações.

1.1 Justificativa

Entender o lado assíncrono de engenharia de software se torna uma tarefa fundamental para o desenvolvimento de novas ferramentas. Para isto, analisar a frequência de uso de funções assíncronas dos módulos core pode oferecer um rico conhecimento para este tema. Pensando neste propósito, os testes automatizados são uma fonte de dados, pois por meio destes é possível compreender vários cenários de um sistema de forma rápida e eficiente.

O estudo de bugs de *event race* do Node.js (Endo e Møller, 2020), constitui a base deste trabalho, onde este busca acrescentar novos dados neste tema. Estes bugs são caracterizados quando duas ou mais funções assíncronas são executadas paralelamente e o comportamento do sistema depende da ordem de conclusão delas. Por fim, entender a

frequência de uso e principais padrões das funções assíncronas dos módulos core contribui para a compreensão dos possíveis bugs em aplicações reais.

Para atender essa necessidade, este trabalho propõe analisar as funções assíncronas mais utilizadas dos módulos core do Node.js, além de seus padrões assíncronos mais usados em aplicações reais. Com esta finalidade, serão gerados dados em forma de gráficos para compreender como os sistemas utilizam a parte assíncrona do Node.js.

1.2 Objetivos

O objetivo deste trabalho foi examinar projetos que utilizam como base o JavaScript (Node.js), nos quais buscou-se entender e analisar a frequência de uso das funções assíncronas dos módulos core do Node.js. O intuito era verificar quais módulos, funções e padrões assíncronas mais utilizados, além de realizar paralelos entre esses dados e situações comuns no desenvolvimento de aplicações Node.js.

1.3 Estrutura do Trabalho

Este trabalho está dividido na seguinte forma: o Capítulo 2 apresenta o conceito de teste automatizados, Node.js e seus módulos core, além de enunciar os trabalhos relacionados; o Capítulo 3 apresenta a metodologia utilizada para a condução do estudo; no Capítulo 4 têm-se a apresentação e análise dos resultados; por fim, no Capítulo 5 as conclusões a respeito do que foi estudado neste trabalho são apresentadas.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentados os conceitos e as principais ferramentas de testes automatizados em aplicações Node.js. Portanto, o capítulo é iniciado com os conceitos básicos de testes de software na Seção 2.1, os principais *frameworks* de teste em aplicações Node.js na Seção 2.2 e, por fim, os trabalhos relacionados na Seção 2.3.

2.1 Conceitos básicos de Teste de Software

Ao longo dos anos do desenvolvimento de aplicações em geral, cada vez mais cresce a atenção para testes automatizados. Considerando isso, hoje pode ser afirmado que um sistema sem testes automatizados é um sistema vulnerável a erros, com significativas dificuldades na realização de manutenções e implementação de evoluções.

Para testar um trecho de código, é criado um caso de teste, sendo seu intuito é de validar um comportamento esperado a partir de uma entrada de dados. Os casos de teste têm uma descrição breve sobre o que ele deve fazer e seus critérios de sucesso.

Um empecilho associado com os casos de teste de execução manual é a necessidade de repetidas execuções. Isso resulta em um consumo de tempo desnecessário, especialmente ao testar cenários relativamente simples. Para solucionar isso, foram criados os testes automatizados, que a partir de um script, testam o trecho com sua devida validação, solucionando o problema da repetição de testes e trazendo eficiência e precisão para o tratamento de cenários mais complexos.

Testes automatizados têm o propósito de verificar e validar o comportamento de um trecho de código, este podendo ser chamado de *System Under Test* (SUT). Na Figura 1, é apresentado um exemplo de um teste automatizado usando Node.js e o *framework* Jest. Nele, o intuito é obter as informações de um usuário. Para isto, a linha 1 descreve o objetivo do teste e as linhas 2 até 8 é feita a criação de um usuário, sendo utilizado o conceito de *InMemoryTestDatabase* (Fowler, 2005), um padrão que substitui um banco de dado convencional por um banco simulado em memória. Linhas 10 até 11 executa o SUT e nas linhas 14 e 15 a validação dos dados recebidos.

Figura 1: Exemplo de teste para pegar as informações de um usuário

```
1   it('should be able to get user profile', async () => {
2       const userCreated = await inMemoryUserRepository.
3       create({
4           name: 'JohnDoe',
5           email: 'johndoe@example.com',
6           birth: new Date(2000, 0, 12),
7           password_hash: await hash('123456', 6)
8       });
9
10      const { user } = await sut.execute({
11          userId: userCreated.id
12      });
13
14      expect(user.id).toEqual(expect.any(String));
15      expect(user.email).toEqual('johndoe@example.com');
16  });
```

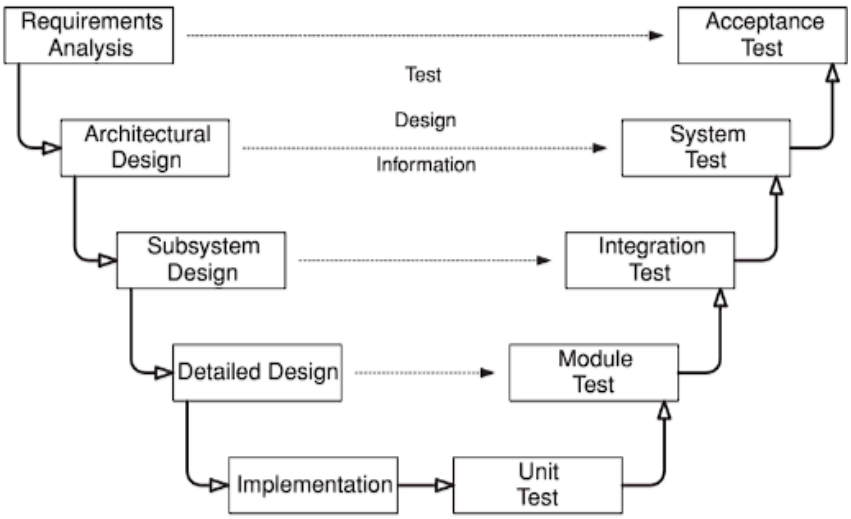
Fonte: Autoria própria

Segundo Ammann e Offutt (2008), é possível categorizar os testes em cinco categorias:

- Teste unitário: avaliar se um trecho unitário (classe, função) tem um comportamento esperado.
- Teste de módulos: avaliar se um módulo isolado funciona corretamente, incluindo seus componentes unitários.
- Teste de integração: avaliar se os módulos do sistema funcionam corretamente juntos.
- Teste de sistema: avaliar se o sistema, como um todo, atende aos requisitos funcionais e não funcionais.
- Teste de aceitação: avaliar se o sistema, como um todo, atende às necessidades do usuário final.

A Figura 2 apresenta o "V model", um modelo que apresenta uma visão de criação de testes, desde o nível de análise dos requisitos até a implementação. Com esse modelo, é mostrada uma estratégia de desenvolvimento de software em paralelo com testes automatizados, onde desde a implementação de um requisito é realizado seu teste, aumentando a robustez do sistema desde o início do desenvolvimento.

Figura 2: V Model - Desenvolvimento de software e níveis de teste



Fonte: Ammann e Offutt (2008)

2.2 Node.js e Frameworks de teste

Node.js é uma plataforma que permite a utilização de JavaScript no lado servidor, sendo baseado na engine V8 desenvolvido pelo Google. A engine é utilizada no navegador Google Chrome, sendo capaz de compilar e executar código fonte JavaScript (V8, 2024). O Node.js tem como principal característica a execução de forma *single-thread*, então as funções assíncronas não bloqueiam a *thread* principal. Portanto, esse ambiente oferece um fácil controle das funções bloqueantes, sendo plausível a utilização dele para desenvolver sistemas escalonáveis (Node.js, 2023).

O Node.js vem acompanhado de alguns pacotes para o seu funcionamento, esses são conhecidos como módulos core. Estes módulos são essenciais para o funcionamento do Node.js e suas bibliotecas, fornecendo uma série de recursos, como manipulação de arquivos, operações de entrada e saída (I/O), gerenciamento de rede, criptografia e muitos outros. Os módulos core podem ser importados como qualquer outra biblioteca, mas o Node.js permite que seja possível colocar um prefixo *node:* para esses módulos serem facilmente reconhecidos. A Figura 3 demonstra uma função assíncrona do Node.js, no qual será gerado dezesseis *bytes* de forma aleatória. Após ser completa, será chamada a função *callback*, que caracteriza um tipo de padrão assíncrono.

Figura 3: Exemplo de função assíncrona do Node.js

```
1       import crypto from 'node:crypto'
2
3       crypto.randomBytes(16, function callback() {
4           // todo
5       })
```

Fonte: Autoria própria

Em Node.js, existem vários *frameworks* focados em testes. O próprio possui módulos feitos para tal, entretanto, algumas funcionalidades ainda estão em estado experimental. Os *frameworks* mais populares são o Jest e Mocha (StateOfJavaScript, 2022), portanto esta seção irá focar apenas nestes dois.

Jest é um *framework* desenvolvido pelo Facebook e pode ser descrito como: "Um *framework* de teste em JavaScript projetado para garantir a correção de qualquer código JavaScript. Ele permite que você escreva testes com uma API acessível, familiar e rica em recursos que lhe dá resultados rapidamente" (Jest, 2023). Portanto, o Jest é uma ferramenta de teste para desenvolvimento com Node.js, se destacando por sua simplicidade de código e isolamento de testes. Ele assegura que um teste não influencie outro, garantindo que cada teste seja executado em um ambiente isolado e limpo. Entretanto, essa funcionalidade de isolamento não afeta o banco de dados. Na Figura 1 foi apresentado um exemplo de código feito com Jest.

Mocha é um *framework* desenvolvido pela Localize e é definido da seguinte forma: "Mocha é um *framework* de teste JavaScript rica em recursos executada em Node.js e no navegador, tornando os testes assíncronos simples e divertidos. Os testes Mocha são executados em série, permitindo relatórios flexíveis e precisos, enquanto mapeiam exceções não detectadas para os casos de teste corretos" (Mocha, 2023). Portanto, o Mocha tem como principais características a possibilidade de realizar testes em navegadores e de forma sequencial.

Existem diferenças entre os dois *frameworks*, a escolha do mais adequado depende do contexto e das necessidades do projeto. Mocha é mais flexível, pois possui capacidade de suportar uma ampla variedade de bibliotecas e estilos de teste, portanto ideal em cenários no qual é necessário um grande número de configurações de bibliotecas. Enquanto o Jest é mais veloz, pois consegue rodar os testes de forma paralela, além de sua simplicidade por sua rápida configuração. Dessa forma, os dois têm suas vantagens e desvantagens, justificando serem os dois *frameworks* mais populares no ambiente de Node.js.

2.3 Trabalhos Relacionados

Ganji, Alimadadi e Tip (2023) defendem em sua pesquisa que os critérios tradicionais de cobertura de código não refletem a qualidade de um conjunto de teste, quando estes

possuem comportamentos assíncronos. Para adequar-se a esta realidade, os pesquisadores apresentaram três novos critérios nestas coberturas. O primeiro critério se refere a conclusão de operações assíncronas, lidando com casos de sucessos e exceções. O segundo registra os resultados do primeiro critério, para lidar com ambos resultados possíveis. Por último, são validados os resultados através de testes. Com estes critérios definidos, os autores apresentam a ferramenta JScope, sendo útil para medir automaticamente a cobertura de código em aplicações JavaScript.

Wang et al. (2017) expõem alguns casos de bugs que ocorrem no Node.js, principalmente se tratando de operações assíncronas. A pesquisa tem como intuito apresentar bugs de concorrência e as raízes dos problemas, além de possíveis soluções. Três tipos de violações são estudados, sendo eles: violação de ordem, atomicidade e *starvation*. No artigo, é apresentado o NodeCB, um estudo que analisa cinquenta e sete bugs e seus resultados. A abordagem tem o intuito de descobrir e ajudar na correção de bugs de concorrência no futuro.

Endo e Møller (2020) aprofundam-se na análise dos bugs de *event race*, criando a abordagem chamada NodeRacer, que é composta por três fases. A primeira fase tem a tarefa de iniciar a aplicação e coletar informações importantes para as seguintes fases, que podem ser através dos testes automatizados ou interação manual com a aplicação. A segunda fase utiliza as informações coletadas para inferir relações de *happens-before* entre as *callbacks*, onde esse termo se refere que uma função A deve acontecer antes de uma função B. Por fim, a terceira fase é responsável por executar as aplicações várias vezes adiando as callbacks seletivamente, respeitando a relação de *happens-before*, para expor bugs. A avaliação experimental do NodeRacer mostra que ele revela bugs de *event race* com maior probabilidade e com menos execuções.

Zhou et al. (2023) apresentam uma nova abordagem chamada NodeRT, com foco em simplificar as regras de *happens-before*, para tornar a detecção de bugs de *event race* mais eficientes e práticas. Com base nos dados coletados, esta abordagem analisa os acessos simultâneos a recursos compartilhados, com o objetivo de identificar situações potenciais dos bugs. Por fim, a abordagem ainda tem foco em eliminar os falsos positivos, isto envolve uma filtragem para evitar alarmes desnecessários. Com os resultados, a abordagem reconheceu todos os tipos de bugs conhecidos e nove casos novos em aplicações reais.

2.4 Considerações Finais

Foi possível identificar que existem trabalhos que visam encontrar bugs de *event race*. Estes bugs são frequentemente causados por funções assíncronas, pois seu comportamento exige um cuidado maior do desenvolvedor na implementação. Para entender esses usos, este estudo se dedica a analisar e apresentar os módulos, funções e padrões assíncronas

mais utilizados em aplicações reais, pois com estes, é possível entender alguns comportamentos no desenvolvimento de aplicações e frequência de uso dessas ações assíncronas.

3 METODOLOGIA DO ESTUDO

Este capítulo apresenta o passo-a-passo da realização do estudo proposto. Como o objetivo deste trabalho foi estudar a frequência dos módulos assíncronos em projetos Node.js, foram formuladas as seguintes Questões de Pesquisa (QP):

- **QP1 - Qual a distribuição de uso de módulos assíncronos por projeto?**: o intuito desta questão é identificar como que acontece a distribuição de uso de módulos assíncronos por cada projeto;
- **QP2 - Qual a proporção de uso de módulos assíncronos por projeto?**: a proposta desta questão é identificar a proporção dos módulos assíncronos em cada projeto;
- **QP3 - Qual o padrão de comportamento assíncrono mais utilizado?**: nesta questão buscou-se o padrão mais utilizado em todos os projetos;
- **QP4 - Quais as funções assíncronas mais utilizadas de maneira global e por módulo?**: esta questão de pesquisa foca na identificação das funções assíncronas mais utilizadas nos projetos, tanto em uma perspectiva global quanto em um contexto de módulo.

Desta forma, o capítulo fica estruturado da seguinte maneira: na Seção 3.1 apresenta os padrões assíncronos do Node.js, na Seção 3.2 detalhou-se sobre o protótipo usado para gerar os registros das funções assíncronas em cada projeto, a Seção 3.3 mostra como foi desenvolvida a ferramenta para gerar os dados, a Seção 3.4 fala sobre as escolhas dos projetos. Por fim, na Seção 3.5 descreveu-se possíveis ameaças à validade do trabalho proposto e as considerações finais.

3.1 Os padrões assíncronos do Node.js

Como um passo inicial do desenvolvimento deste projeto, buscou-se entender os padrões assíncronos do Node.js. Sendo definido da seguinte maneira:

- **Simple callback (CB)**: Uma callback é uma função que é passada como argumento de outra função e executada após a conclusão desta. Esse é exemplificado na linha 8 da Figura 4;
- **Return object (RO)**: Um objeto que é retornado após a conclusão de uma função, onde esse objeto pertence a uma classe que pode conter funções assíncronas;
- **Object creation (OC)**: Cria um objeto com a diretiva *new*, bastante parecido com o RO, onde ao invés de chamar uma função e receber um objeto, é instanciado

uma classe diretamente pelo seu *construtor*, como mostrado na linhas 10 até 12 da Figura 4;

- **Object property (OP)**: Neste caso, um objeto assíncrono possui uma propriedade que é outro objeto assíncrono. Este objeto assíncrono se refere a um objeto que possui funções assíncronas. Na Figura 4 linhas 14-18 é representado um exemplo deste padrão no qual o usuário pode digitar no console e o programa receber e processar esses dados;

- **Callback object (CO)**: Caso onde o argumento de uma *callback* é um objeto assíncrono. As linhas 20-23 da Figura 4 representa esse padrão, onde no caso é criado um servidor http e para requisição responde com um status HTTP 200 e a mensagem "Hello, World!";

- **Return promise (RP)**: Retorna uma *promise*, que consiste em um objeto que representa o resultado eventual de uma operação assíncrona. Uma chamada bastante conhecida desse padrão é a *promise* do módulo fs com a função de ler arquivos (*readFile*), ela é representada na linha 25 da Figura 4;

- **Direct delay (DD)**: Um *delay* é aplicado diretamente na função, onde o comportamento padrão aguarda o fim desse tempo para ser executado novamente. O contexto de aplicação desse padrão é em funções de *transform*, onde a classe vai receber uma parte dos dados totais e transformar em novos, assim colocando um delay entre receber novos dados e a espera pelo processamentos.

Figura 4: Exemplo dos padrões assíncronos do Node.js

```
import fs, { promises as fsPromises } from 'node:fs'
import process from 'node:process';
import http from 'node:http';
import net from 'node:net';
import crypto from 'node:crypto';
import stream from 'node:stream';

crypto.randomBytes(16, callback)

const socket = new net.Socket()

socket.on('connect', () => {...})

const process_stdin = process.stdin

process_stdin.on('data', (data) => {
    console.log(`Dado fornecido: ${data}`)
})

const server = http.createServer((req, res) => {
    res.writeHead(200)
    res.end('Hello, world!')
})

fsPromises.readFile('/dir/to/file', 'utf-8').then( function callback() {
    //todo
} );
```

Fonte: Autoria própria

3.2 Protótipo NACD

O protótipo NACD, desenvolvido pelo grupo de pesquisa do Prof. André Takeshi Endo, é uma ferramenta para detectar bugs de condições de corrida em aplicações Node.js. A abordagem adotada pelo NACD consiste na estratégia de injetar um *delay* nas funções assíncronas dos módulos core do Node.js. Para tal, foi realizada uma análise das funções e objetos assíncronos presentes no Node.js. Com base nesta análise, é construído um modelo para ser manipulado no protótipo.

O NACD fornece vários tipos de registros das funções assíncronas em cada teste. Para este trabalho, um tipo de registro é usado, seguindo o modelo da Figura 5. Neste registro de log, os dois campos consistem, respectivamente, no nome do módulo e sua função usada. A partir desses registros, serão gerados novos dados para uma análise das QPs.

Figura 5: Exemplo dos registros gerados pelo NACD

```
1    {
2      ...
3      "object": "fs",
4      "function": "stat",
5      ...
6    }
```

Fonte: Disponibilizado pelo grupo de pesquisa do Prof. André Takeshi Endo

3.3 Ferramenta para gerar os dados

Esta ferramenta foi criada no contexto deste trabalho, sendo desenvolvida com Node.js e TypeScript. Este projeto possui 2749 linhas de código e está disponibilizado como um projeto *open source* <https://github.com/Julio-Cesar07/tcc>.

Os registros gerados pelo protótipo *NACD* permitem uma análise detalhada dos projetos, possibilitando a filtragem específica por módulos, funções e padrões utilizados nos testes dos projetos. Para alcançá-los, foi desenvolvida uma ferramenta que analisa os registros em JSON e gera dois tipos de tabelas: a primeira apresenta os dados de cada módulo e os padrões mais utilizados, separados por projetos; a segunda é uma divisão em nível de módulos, no qual são contabilizadas as funções mais frequentemente utilizadas.

Para o desenvolvimento desta ferramenta, alguns pontos são necessários. Um módulo core possui uma ou mais classes, então a ideia da ferramenta é devolver os padrões correspondentes de acordo com o módulo fornecido e sua função. As funções podem conter mais de um padrão assíncrono, então é retornado um vetor. Por fim, deve ser contabilizado no campo do módulo desconsiderando a classe, então uma função do http.Server deve ser contabilizada no módulo http.

No desenvolvimento desta ferramenta, inicialmente foram mapeados as classes dos módulos com suas funções e respectivos padrões assíncronos. Os registros do NACD nos fornecem o nome da função e seu módulo. Por exemplo, na Figura 6 são apresentadas as funções da classe *ClientRequest* do módulo *http*, no qual é realizado um *switch case* que recebe como argumento o nome da função e retorna o seu padrão correspondente. Existem classes de um módulo que são filhas de outras classes; no caso da Figura 6, algumas funções desta classe pertencem à classe pai *EventEmitter*. Portanto, caso a função caia no *default* (linha 7), o nome da função é repassado para o *switch case* do pai, retornando assim o padrão correto. Outra consideração importante é a realização de uma divisão entre classes de um mesmo módulo. Por exemplo, o módulo *http* possui as seguintes classes: *http* (a si próprio), *http.Agent*, *http.ClientRequest*, *http.IncomingMessage*, *http.Server* e *http.ServerResponse*. Portanto, para encontrar as devidas funções, é feito mais um *switch*

case, cujos parâmetros são o nome do módulo com a classe correspondente (*module_name*) e a sua função (*function_name*), sendo detalhado na Figura 7.

Figura 6: Exemplo das classes com suas devidas funções e padrão assíncrono

```
1     function typeHttpClientRequest(function_name: string){
2         switch (function_name) {
3             case 'setTimeout':
4             case 'write':
5             case 'end':
6                 return ['cb'];
7             default:
8                 return typeEventEmitter(function_name)
9         }
10    }
```

Fonte: Autoria própria

Figura 7: Caso de escolha da classe http

```
1     export function chooseHttp(module_name: string,
2         function_name: string) {
3     let types_returns: string[] = []
4
5     switch (module_name) {
6         case 'http':
7             types_returns = typeHttp(function_name);
8             break;
9         case 'http.Agent':
10            types_returns = typeHttpAgent(function_name);
11            break;
12        case 'http.ClientRequest':
13            types_returns = typeHttpClientRequest(function_name);
14            break;
15        case 'http.IncomingMessage':
16            types_returns = typeHttpIncomingMessage(function_name);
17            break;
18        case 'http.Server':
19            types_returns = typeHttpServer(function_name);
20            break;
21        case 'http.ServerResponse':
22            types_returns = typeHttpServerResponse(function_name);
23            break;
24        default:
25            console.log('http not found')
26            console.log(module_name + " " + function_name)
27    }
28
29    return types_returns
30 }
```

Fonte: Autoria própria

Após a criação de todas as classes de todos os módulos core do Node.js, procede-se ao processamento dos registros disponibilizados pelo NACD. A Figura 8 ilustra as colunas e os dados contidos na primeira tabela gerada. Para cada projeto, são contabilizadas a quantidade de funções executadas por um módulo, o total de chamadas dessas funções e seus respectivos padrões assíncronos retornados.

Figura 8: Tipo de dados gerados

```
1       type DataFormat = {
2           project_name: string;
3           total_quantity: number;
4           child_process: number;
5           cluster: number;
6           crypto: number;
7           dgram: number;
8           dns: number;
9           eventEmitter: number;
10          fs: number;
11          http: number;
12          http2: number;
13          https: number;
14          net: number;
15          process: number;
16          readline: number;
17          repl: number;
18          stream: number;
19          tls: number;
20          util: number;
21          zlib: number;
22          cb: number;
23          ro: number;
24          oc: number;
25          op: number;
26          co: number;
27          rp: number;
28          dd: number;
29      }
```

Fonte: Autoria própria

Portanto, o processo da ferramenta envolve acessar cada diretório que contém os registros gerados dos testes de cada projeto, seguido pelo processamento desses dados. Conforme ilustrado nas linha 4-7 da Figura 9, é criado uma variável chamada *data_aux* a qual representa a linha da tabela com os dados processados, sendo composto pelo nome do projeto (*project_name*) e pela função *initDataFormat()* que é retornado um objeto do tipo *DataFormat* (Figura 8) com os valores inicialmente zerados. No tratamento dos registros, a estratégia de *promises* é empregada por meio da função *Promise.all* para permitir a execução paralela das operações, no qual cada registro chama a função *process_files* (linha 18), resultando na quantidade de módulos e padrões utilizados, representado por *aux* na linha 23 da Figura 9. Após a conclusão de todas as *promises*, um vetor do tipo *DataFormat* sem o campo *project_name* é retornado. Em seguida, procede-se com a soma de todos os campos do vetor *aux*, como mostrando nas linhas 26-28 é feita a soma de todos os itens do vetor somando o campo *total_quantity* de cada um, armazenando o resultado em cada campo correspondente do *data_aux*.

Figura 9: Processamento dos diretórios

```
1
2       async function process_dir(pathDir: string,
3       project_name: string){
4           const data_aux: DataFormat = {
5               project_name,
6               ...initDataFormat()
7           }
8
9           try {
10              // processamento dos registros
11              const files = await fsPromises.readdir(pathDir);
12              const promises_process_files = files
13              .map<Promise<Omit<DataFormat, 'project_name'>>>
14              (async file => {
15                  if(file.startsWith('nacd-never')){
16                      const pathFile = path.join(pathDir, file)
17
18                      return process_files(pathFile)
19                  }
20                  throw new Error('files not is nacd-never')
21              })
22
23              const aux = await
24              Promise.all(promises_process_files)
25
26              data_aux.total_quantity = aux.reduce
27              ((total, currentValue) => total +
28              currentValue.total_quantity, 0)
29              ...
30
31              // escrita na tabela
32              const file_csv =
33              fs.createWriteStream(path.join(__dirname,
34              `../data/data.csv`), { flags: 'a'})
35              const headers = [...]
36
37              fastcsv.write([data_aux], { headers,
38              includeEndRowDelimiter: true,
39              writeHeaders: false }).pipe(file_csv)
40
41              ...
42      }
```

Fonte: Autoria própria

O processamento de cada projeto é feito sequencialmente, seria possível ser feitos juntos utilizando as *promises* juntamente com a função *Promise.all*, entretanto, essa estratégia pode extrapolar o limite de memória do aplicativo. A fim de evitar esse problema, a ferramenta processa um projeto por vez e após contabilizar todos os registros, é feito a escrita da linha do projeto na primeira tabela, como apresentado nas linhas 31-38 da Figura 9.

A Figura 10 mostra como é feito os processamentos de cada registro. No qual é iniciado uma constante *data_ aux* com os valores zerados (linha 3), ela representa um registro que será retornado no vetor da função *process_ dir* da Figura 9 (linha 18). Após isso, é feita a leitura do primeiro registro (linhas 5-6). Após a leitura, é criado um vetor com cada registro do NACD (linha 8-12) e caso não seja um vetor, é retornado os valores zerados (linha 14). Para cada *log*, é verificado se ele é uma operação válida (linhas 17-19), caso não

seja, a função ignora e retorna sem nenhum processamento. Caso seja válido, a quantidade total de funções é incrementada (linha 22) e verificado se ele é alguma propriedade de *DataFormat* (linhas 26-38). Caso seja, incrementa a quantidade desse módulo (linha 27 e 30).

Em alguns casos, o protótipo *NACD* pode escrever um *log* com o nome da função fora do padrão, portanto, é feito um ajuste em casos específicos para o módulo e função correto (linhas 28-32). Após as verificações, é chamado uma função *defineModule*, que recebe o nome do módulo (*log.object*) e o nome da função (*log.function*), onde ela seleciona o módulo correto e chama a função apresentada na Figura 7 para retornar os padrões que são incrementados em *data_ aux* (linha 42).

Figura 10: Processamento dos registros

```
1         async function process_files(pathFile: string):
2         Promise<Omit<DataFormat, 'project_name'>> {
3         const data_aux = initDataFormat()
4         try {
5             const data =
6             await fsPromises.readFile(pathFile, 'utf-8');
7
8             const logs: {
9                 object?: string | null,
10                operation?: string | string[]
11                function?: string
12            }[] = JSON.parse(data)
13
14            if(!Array.isArray(logs)) return data_aux
15            logs.forEach(log => {
16                if(!log.operation
17                || log.operation === 'callback run'
18                || log.operation === 'promise run'){
19                    return
20                }
21
22                data_aux.total_quantity++;
23                const key = log.object ? log.object
24                .split('.', 1)[0].toLowerCase() : "";
25
26                if (key in data_aux) {
27                    data_aux[key]++;
28                } else if(key === 'writestream'
29                || key === 'readstream'){
30                    data_aux.fs++;
31                }
32                ... else {
33                    if(log.operation !== 'direct delay')
34                        ...
35                    else {
36                        data_aux.dd++;
37                    }
38                }
39
40                if(log.function && log.object)
41                    defineModule(log.object, log.function)
42                    .forEach(item => data_aux[item]++)
43            })
44            return data_aux;
45            ...
```

Fonte: Autoria própria

Após gerar a primeira tabela, é iniciado o processamento da segunda tabela, que consiste nas funções de cada módulo. Seguindo a mesma ideia discutida anteriormente, são gerados os dados para as funções, que consistem em quatro colunas: a primeira com o nome do módulo, a segunda com o nome da função deste módulo, quantidade de vezes da função chamada e em quantos projetos em que ela foi usada.

O desenvolvimento tem algumas diferenças da primeira tabela, a principal sendo a criação de um objeto global para cada módulo *core* (Figura 11). Estes objetos serão os dados que serão escritos na tabela. A cada função de um módulo encontrado nos registros, é verificado se esta já existe no objeto do seu módulo, caso exista é incrementado no *total_quantity* desta função, caso não é criado uma propriedade com o nome desta função e o *total_quantity* com valor 1. O campo *times_used* consiste em quantos projetos esta função foi usada ao menos uma vez, portanto, para manter a integridade é feita a seguinte verificação no vetor global *modules_functions_in_project* (linha 6, Figura 12): se é a primeira vez que a função aparece neste projeto, é adicionado nesta variável global o nome da função e seu módulo; caso exista, apenas ignora; portanto, esse vetor tem a função de guardar quais funções já foram usadas no projeto. Após processar todos os registros de um projeto, é feita a distribuição do campo *times_used* de cada item do vetor global. Por fim, o vetor global é zerado para o próximo projeto.

Figura 11: *Interface* da tabela de funções

```
1         interface ModuleFunctions {
2             [key: string]: {
3                 total_quantity: number
4                 times_used: number
5             };
6         }
7
8         const childProcessFunctions: ModuleFunctions = {}
```

Fonte: Autoria própria

Figura 12: Vetor global para contabilizar em quantos projetos foi usado uma função

```
1         interface modules_functions {
2             module_name: string;
3             function_name: string;
4         }
5
6         let modules_functions_in_project: modules_functions[]
7         = []
8
9         function reset_modules_functions() {
10            modules_functions_in_project = []
11        }
```

Fonte: Autoria própria

3.4 Atividades realizadas

A fim de identificar projetos de software com extensa cobertura de testes, adotou-se a seguinte metodologia:

- **Seleção de Projetos**: Inicialmente, foram selecionados projetos de duas fontes disponíveis online: Awesome Node.js[1] e Awesome Node.js Projects[2]. Estas fontes listam diversos projetos open-source voltados para a comunidade de desenvolvedores, totalizando 575 projetos;
- **Verificação de Testes**: Para cada um dos 575 projetos, foi verificada a presença de testes automatizados. Utilizou-se o comando *npm run test*, executado através do protótipo NACD, para filtrar os projetos que não tinham testes.
- **Filtragem por Operações Assíncronas**: Após a execução do comando, realizou-se um filtro manualmente para excluir projetos que não possuíam operações assíncronas, pois estas são o alvo desta pesquisa;

Com este procedimento, foi possível restringir o conjunto inicial para 336 projetos, os quais foram utilizados para análises e estão disponíveis em <https://github.com/Julio-Cesar07/tcc/blob/main/data/githubStats-2020-04-08.csv>.

3.5 Ameaças a validade

Uma possível deficiência dos dados gerados é que a ferramenta desenvolvida ou o protótipo NACD podem conter uma implementação incorreta. Entretanto, a cada resultado gerado uma verificação e avaliação dos dados foram feitas, portanto ambas ferramentas foram testadas para serem diminuídos ao máximo esses possíveis erros. Outra possível ameaça é os projetos não serem uma amostra representativa, porém os projetos passaram pelos filtros descritos na Seção 3.4 e validados para esse estudo. Ainda, os dados gerados são dependente da qualidade na implementação dos casos de teste nas aplicações. Além disso, com o avanço do desenvolvimento do Node.js, algumas funções ou até mesmo módulos podem ser substituídos ou aprimorados. Por exemplo, o módulo *http2* não apareceu em qualquer registro, mesmo sendo projetado para ser uma evolução significativa do módulo *http*. Essa deficiência pode limitar a validade das conclusões do estudo.

Após a conclusão do desenvolvimento desta ferramenta, foram obtidos todos os dados já refinados com relação ao estudo. Como as análises ocorreram de acordo com as questões de pesquisa, todos os gráficos ou tabelas gerados e coletados durante as etapas citadas, foram agrupados e discutidos no próximo capítulo, bem como a discussão dos resultados.

[1] <https://github.com/sindresorhus/awesome-nodejs>
[2] <https://github.com/sqreen/awesome-nodejs-projects>

4 ANÁLISE DE RESULTADOS

Neste capítulo serão apresentados os resultados obtidos durante a execução do estudo citado no capítulo anterior. Serão realizadas também as discussões e análises destes. O capítulo ficou estruturado da seguinte maneira: a Seção 4.1 analisa em detalhes os projetos selecionados, nas Seções 4.2 a 4.5 discorreu-se mais em detalhes sobre os dados colhidos visando responder às questões de pesquisas propostas por este trabalho.

4.1 Caracterização dos projetos selecionados

Após a filtragem dos projetos, detalhados na Seção 3.4, as 336 aplicações restantes são caracterizadas na Tabela 1. Para identificar o nível de popularidade dos projetos, foram selecionados o número de estrelas, *forks* e *open issues* de cada projeto no GitHub.

Tabela 1: Dados dos projetos analisados

	Stars	Forks	Open Issues
máximo	71624	8976	2091
mínimo	10	1	0
mediana	1907	157	20
média	6027,88	642,23	86,52

Fonte: Autoria própria

Nessa base de projetos, a aplicação com mais estrela possui um total de 71.624 estrelas, além do menor com apenas 10. Ainda em relação ao número de estrelas, a média dos projetos constitui um valor de 6.027,88 estrelas e uma mediana com 1907. O máximo de *forks* e *open issues* foram, respectivamente, 8.976 e 2.091, com mínimo de apenas 1 *fork* e nenhum *open issues*.

A Tabela 2 ilustra os três projetos com maior quantidade de estrelas.

Tabela 2: Projetos com mais estrelas

Name	Stars
mzabriskie_axios	71624
GoogleChrome_puppeteer	60021
webpack_webpack	53679

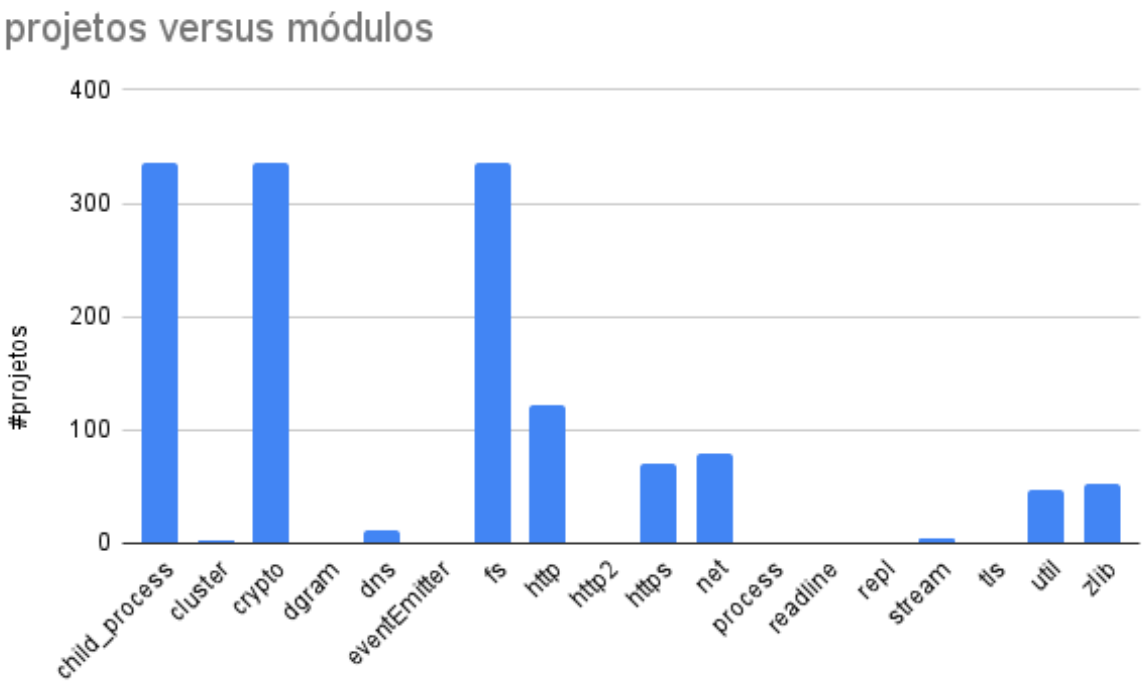
Fonte: Autoria própria

O Axios é uma ferramenta que facilita o envio de solicitações HTTP, como GET, POST, PUT, DELETE, entre outras. O Puppeteer permite a automatização de ações em navegadores Chrome e Chromium. Por fim, o Webpack é um empacotador de módulos, no qual sua função é agrupar arquivos JavaScript para uso em navegador.

4.2 QP 1 - Qual a distribuição de uso de módulos assíncronos por projeto?

Quando analisados os dados gerados, foi possível observar a quantidade de projetos que usam pelo menos uma vez cada módulo em seus testes, isto demonstra a frequência dos módulos nas aplicações analisadas; estes estão enunciados no Figura 13. Neste Gráfico, o eixo horizontal (eixo X) representa os 18 módulos assíncronos do Node.js, sendo cada módulo representado por uma coluna individual. No eixo vertical (eixo Y), é mostrada a quantidade de projetos, sendo o valor mínimo 0 e máximo 336. As colunas representam quantos projetos usaram ao menos uma vez o módulo assíncrono.

Figura 13: Distribuição de uso de módulos assíncronos por projetos



Fonte: Autoria própria

Primeiramente, os módulos *child_process, crypto e fs* são usados em todos os projetos analisados. Isto pode ser explicado pela finalidade de cada módulo, que são essenciais em aplicações desenvolvidas com Node.js. O *child_process* é essencial para manipulação de processos secundários, permitindo que a aplicação interaja com o sistema operacional e execute tarefas paralelas. O módulo *crypto* disponibiliza funções de criptografia e segurança, sendo essenciais para ambientes web. O *fs* permite a manipulação de operações de entrada e saída (I/O), sendo muito útil para importar grandes dados de armazenamentos físicos.

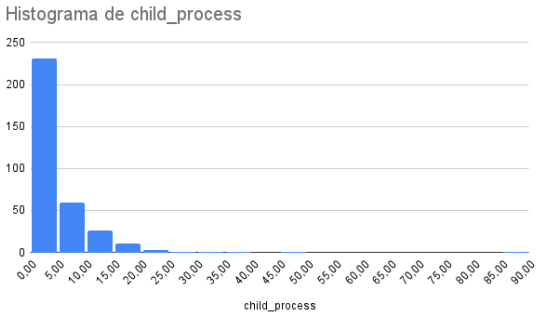
Outra análise é com relação aos módulos não usados. A razão de alguns módulos não

serem utilizados pode se dar pelo motivo deles serem mais frequentemente encontrados em cenários específicos. Por exemplo, os módulos *dgram, readline e repl* são usados em situação mais específicas, sendo elas, respectivamente: em casos de comunicação UDP (que é menos usado que o TCP), leitura de dados do usuário via linha de comando e criação de consoles interativos. No caso dos módulos *EventEmitter, process e tls* uma das possíveis causas deles não serem usados é pelo fato da utilização de *frameworks* e bibliotecas que abstraem seu uso. Além de que eles podem ser usados indiretamente por outros módulos. Por exemplo, o *socket* do *net* implementa um *EventEmitter*, mas esse dado será contabilizado no módulo *net*. Quanto ao *http2*, um possível motivo é o fato dos testes das aplicações ainda usarem o *http*, sendo necessário tempo para a migração entre os dois módulos.

4.3 QP 2 - Qual a proporção de uso de módulos assíncronos por projeto?

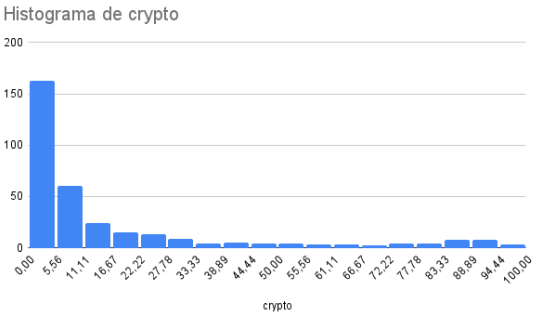
Considerando que os dados brutos às vezes não refletem quanto cada módulo é realmente usado, transformou-se os dados em porcentagem. Para isto, foram criados novos gráficos utilizando o número de vezes em que uma função de um módulo é usado, multiplicado por cem e dividido pelo total de operações assíncronas em cada projeto. Dados estes cálculos, os gráficos que não tiveram chamada foram retirados, pois seus dados são todos zeros, são estes: *dgram, eventEmitter, http2, process, readline, repl e tls*. Os gráficos foram feitos na forma de histograma, no qual o eixo vertical (eixo Y) indica a quantidade de projetos e o eixo horizontal (eixo X) representa a porcentagem de uso, sendo as faixas agrupadas em intervalos; cada histograma é apresentado nas Figuras 14 à 24.

Figura 14: Proporção de uso do módulo child_process por projeto



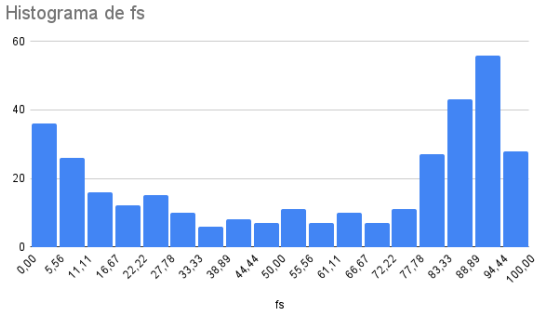
Fonte: Autoria própria

Figura 15: Proporção de uso do módulo crypto por projeto



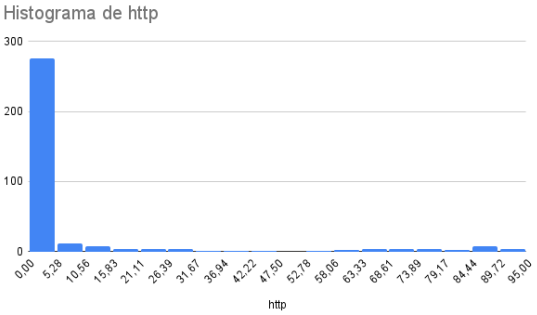
Fonte: Autoria própria

Figura 16: Proporção de uso do módulo fs por projeto



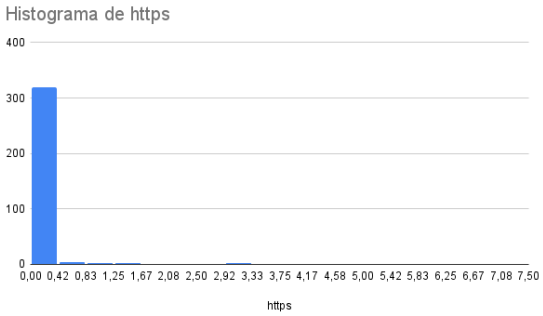
Fonte: Autoria própria

Figura 17: Proporção de uso do módulo http por projeto



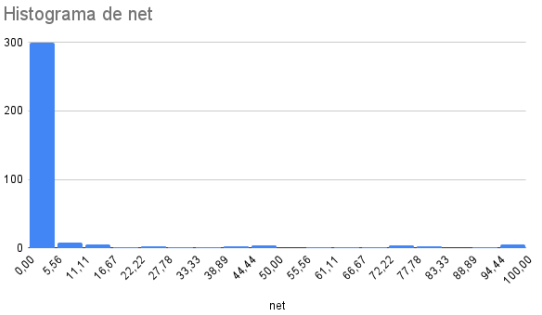
Fonte: Autoria própria

Figura 18: Proporção de uso do módulo https por projeto



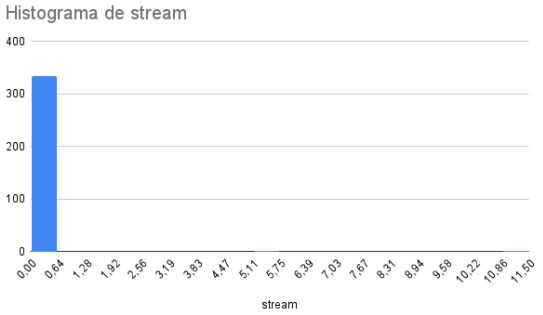
Fonte: Autoria própria

Figura 19: Proporção de uso do módulo net por projeto



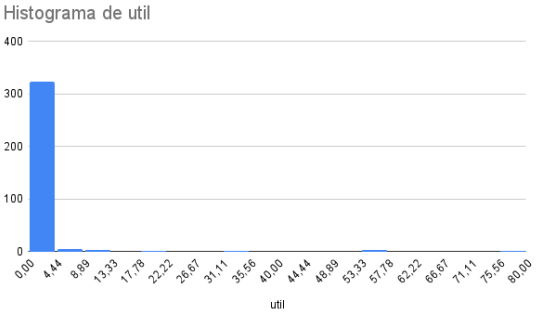
Fonte: Autoria própria

Figura 20: Proporção de uso do módulo stream por projeto



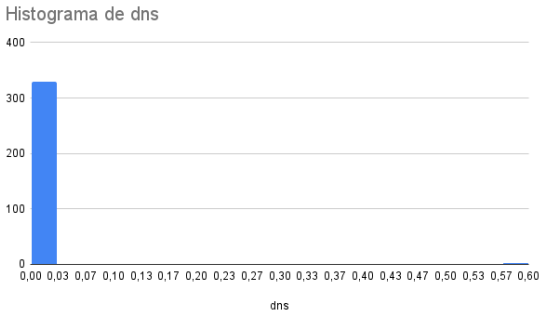
Fonte: Autoria própria

Figura 21: Proporção de uso do módulo util por projeto



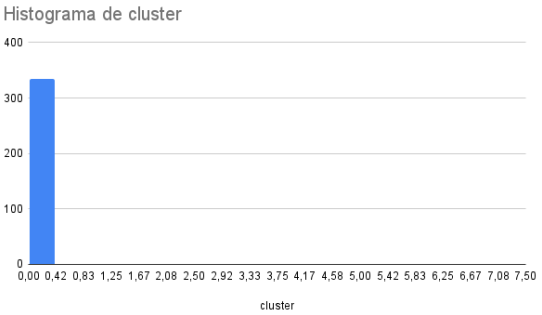
Fonte: Autoria própria

Figura 22: Proporção de uso do módulo dns por projeto



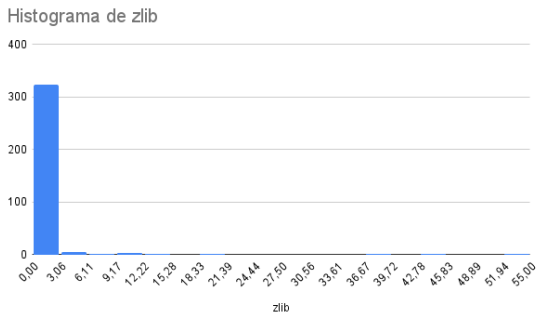
Fonte: Autoria própria

Figura 23: Proporção de uso do módulo cluster por projeto



Fonte: Autoria própria

Figura 24: Proporção de uso do módulo zlib por projeto



Fonte: Autoria própria

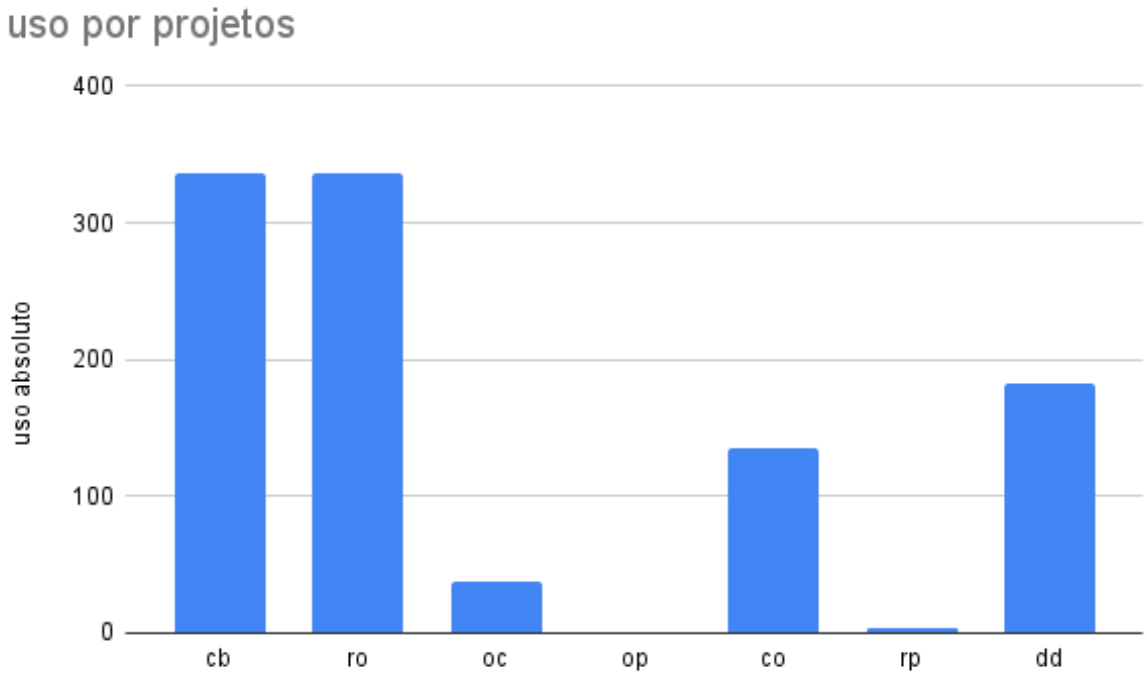
Analisando os histogramas, é possível observar que o módulo *fs* (Figura 16) é o mais utilizado, ocorrendo mais de 70 projetos em que mais de 88,9% das funções assíncronas são do *fs*. Isto demonstra que em vários projetos o *fs* é uma parte central e fundamental nos casos de teste. Em segundo lugar em termos de uso, encontra-se o módulo *crypto* (Figura 15), cujo histograma revela uma maior variedade de uso nas aplicações, em relação ao outros módulos.

Além disso, uma observação importante, a partir dos histogramas, é a presença significativa de um módulo que representa menos de 5,56% das chamadas assíncronas de um projeto (Figuras 18 até 24). Isso sugere que a maioria dos projetos utiliza as funções dos módulos de maneira diversificada. Entretanto, é importante salientar que mais de 100 projetos utilizam ao menos 88,9% de chamadas assíncronas de apenas um módulo, representando praticamente um terço das aplicações em que apenas um módulo é muito dominante.

4.4 QP 3 - Qual o padrão de comportamento assíncrono mais utilizado?

Na terceira questão de pesquisa, é apresentado como são utilizados os padrões nos casos de teste. Durante a Seção 3.1 foi discutido cada padrão; nas Figura 25 e 26 é apresentada como eles são distribuídos em uso por projetos e absoluto, respectivamente. Compreender os padrões predominantes de programação assíncrona é fundamental para entender o comportamento das funções assíncronas no ecossistema Node.js, isto envolve analisar a abordagem mais comum adotada para gerenciar o retorno das chamadas assíncronas.

Figura 25: Padrões mais usados por projeto



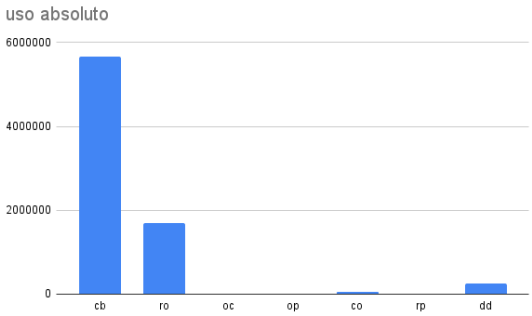
Fonte: Autoria própria

É possível concluir que os padrões mais usados por projeto são *simple callback e return object* (Figura 25). Ambos são bem comuns no desenvolvimento de aplicações Node.js, pois são a forma mais comum dos desenvolvedores lidar com o retorno de uma função assíncrona. O tipo *simple callback* é bastante utilizado pelo fato dos módulos tradicionalmente implementarem as funções com uma *callback*. Enquanto *return object* representa um objeto que contém funções assíncronas, pelo fato dos módulos cores utilizarem orientação objeto, é comum o uso deste padrão.

O terceiro mais utilizado foi o *direct delay*, uma possível explicação é a utilização de um método *transform* para alterar dados. Esta função pode ser chamada em um contexto para transformar um arquivo binário em outro tipo de dado, sendo usado em vários casos de teste. O *callback object* é bastante utilizado na criação de servidores HTTP, portanto bastante comum para responder requisições. O baixo uso do *object creation* pode ser pelo fato dele ser parecido com o ro.

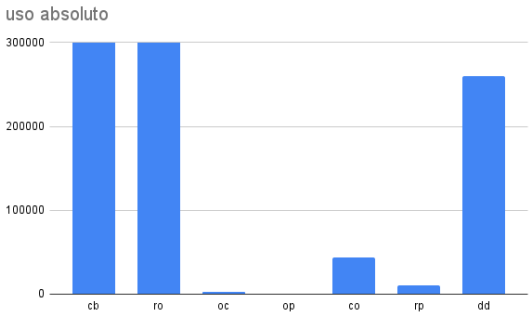
Em relação aos menos usados, o padrão *object property e return promise* estão em certa discrepância comparado aos demais, sendo o primeiro não aparecendo nenhuma vez. Uma possível razão para o baixo uso do *return promise* é pelo fato das *promises* serem implementadas nos módulos cores recentemente, portanto o uso dos outros padrões é mais comum. Enquanto para o *object property* a razão pode ser pelo fato do pouco contato desses padrões em aplicações reais, sendo pouco comum no desenvolvimento.

Figura 26: Padrões mais usados uso absoluto



Fonte: Autoria própria

Figura 27: Padrões mais usados uso absoluto, limite vertical



Fonte: Autoria própria

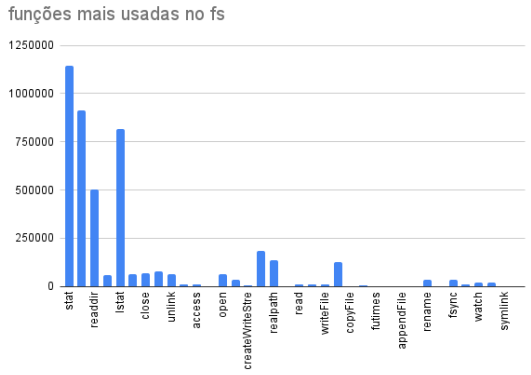
Quanto ao caso de uso absolutos dos padrões (Figura 26), ocorre uma grande quantidade do padrão cb, seguido do ro, demonstrando o motivo de ambos estarem em todos os projetos. A Figura 27 representa a Figura 26 com um limite vertical, para ser possível a visualização dos padrões com menos uso. Os padrões *return promise e object creation* aparecem em menor quantidade no gráfico, sendo o padrão *object property* o único que não apareceu nenhuma vez. Além do motivo descrito anteriormente, isto também pode ser explicado pelo fato deste padrão ter poucas aparições nas funções assíncronas. Como exemplificado na Figura 6, poucas funções tinham como seu retorno o padrão op.

Portanto, grande quantidade das funções assíncronas tem um padrão concentrado entre o cb e o ro, comumente são bastante utilizados no desenvolvimento de aplicações, principalmente em chamadas de API ou respostas do banco de dados.

4.5 QP 4 - Quais as funções assíncronas mais utilizadas de maneira global e por módulo?

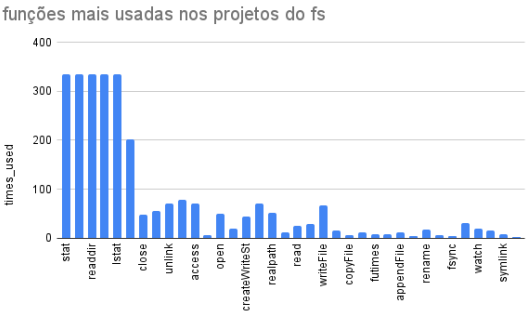
Uma última análise consiste no escopo de funções de módulos, analisando as mais usadas por módulos e as 10 mais usadas de maneira global. As Figuras 28 até 49 demonstram as funções de cada módulos mais utilizadas, tanto em números absolutos quanto em projetos que a utilizam ao menos uma vez. O eixo vertical (eixo Y) representa a quantidade de vezes em que uma função é usada, tanto em projetos quanto em uso absoluto. O eixo horizontal (eixo X) representa o nome da função, sendo cada coluna individual a quantidade de uso daquela função.

Figura 28: Funções assíncronas mais utilizadas do fs



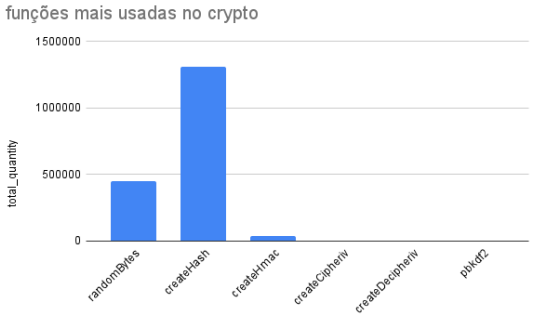
Fonte: Autoria própria

Figura 29: Funções assíncronas mais utilizadas do fs por projetos



Fonte: Autoria própria

Figura 30: Funções assíncronas mais utilizadas do crypto

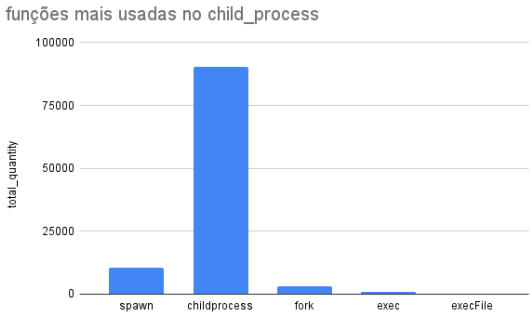


Fonte: Autoria própria

Figura 31: Funções assíncronas mais utilizadas do crypto por projetos

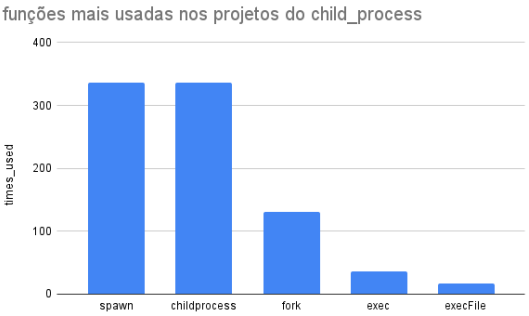
Fonte: Autoria própria

Figura 32: Funções assíncronas mais utilizadas do child_process



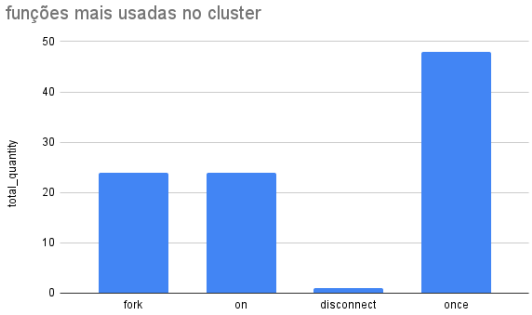
Fonte: Autoria própria

Figura 33: Funções assíncronas mais utilizadas do child_process por projetos



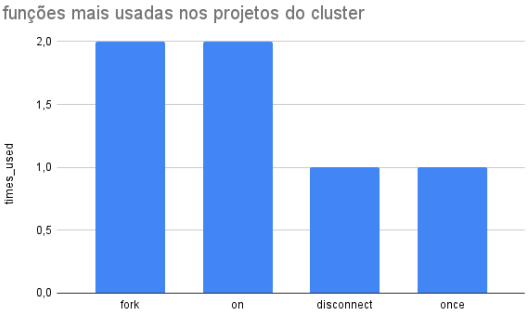
Fonte: Autoria própria

Figura 34: Funções assíncronas mais utilizadas do cluster



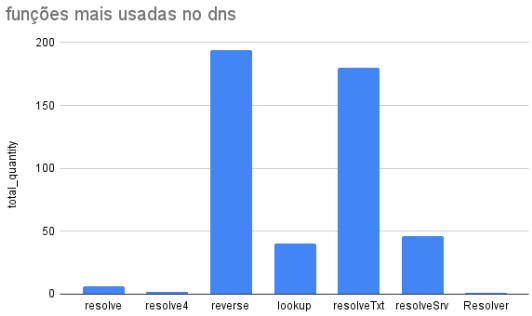
Fonte: Autoria própria

Figura 35: Funções assíncronas mais utilizadas do cluster por projetos



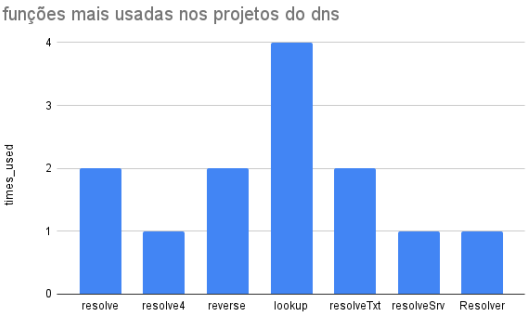
Fonte: Autoria própria

Figura 36: Funções assíncronas mais utilizadas do dns



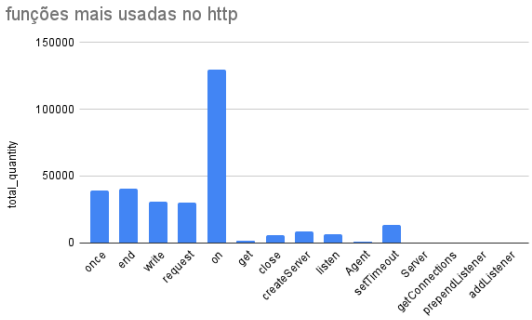
Fonte: Autoria própria

Figura 37: Funções assíncronas mais utilizadas do dns por projetos



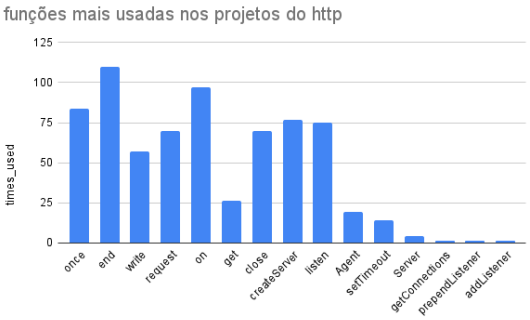
Fonte: Autoria própria

Figura 38: Funções assíncronas mais utilizadas do http



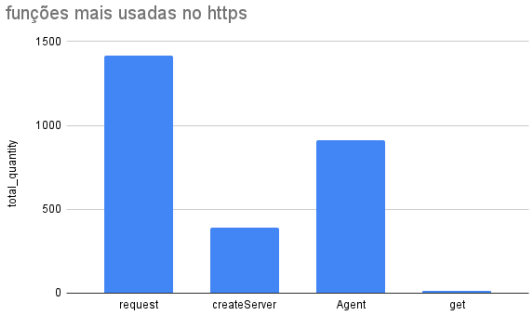
Fonte: Autoria própria

Figura 39: Funções assíncronas mais utilizadas do http por projetos



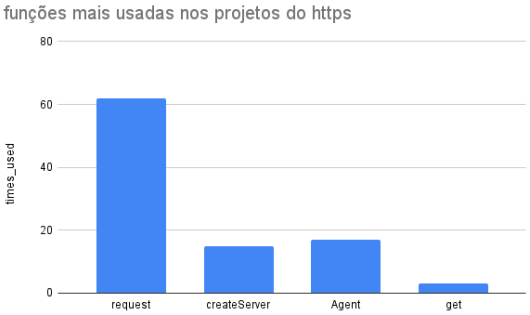
Fonte: Autoria própria

Figura 40: Funções assíncronas mais utilizadas do https



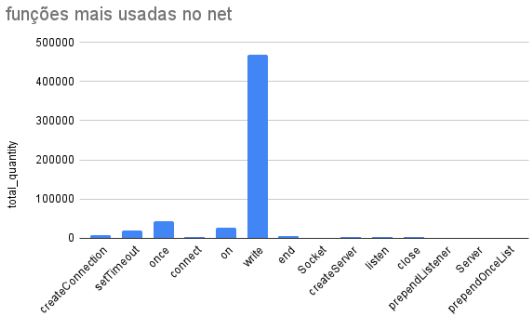
Fonte: Autoria própria

Figura 41: Funções assíncronas mais utilizadas do https por projetos



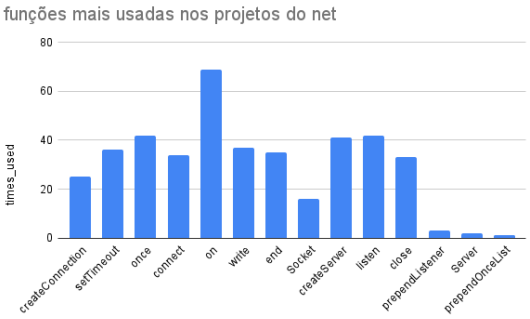
Fonte: Autoria própria

Figura 42: Funções assíncronas mais utilizadas do net



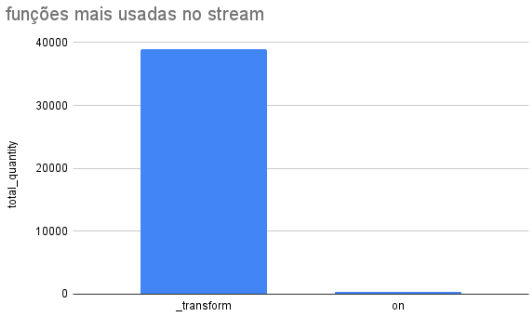
Fonte: Autoria própria

Figura 43: Funções assíncronas mais utilizadas do net por projetos



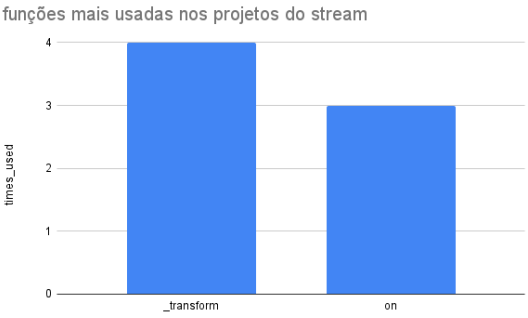
Fonte: Autoria própria

Figura 44: Funções assíncronas mais utilizadas do stream



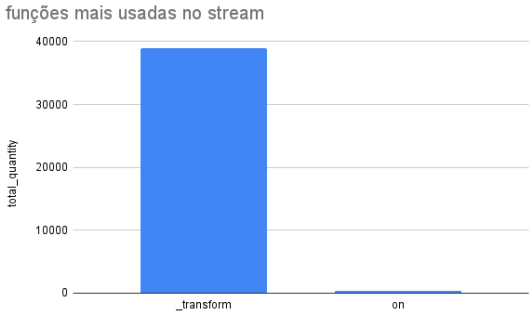
Fonte: Autoria própria

Figura 45: Funções assíncronas mais utilizadas do stream por projetos



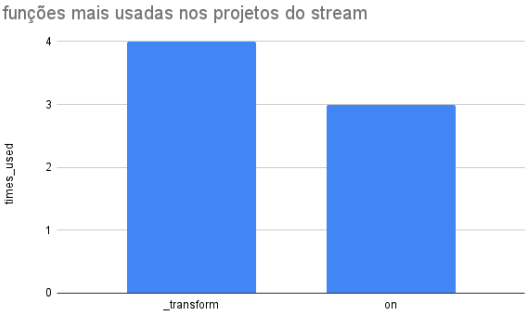
Fonte: Autoria própria

Figura 46: Funções assíncronas mais utilizadas do stream



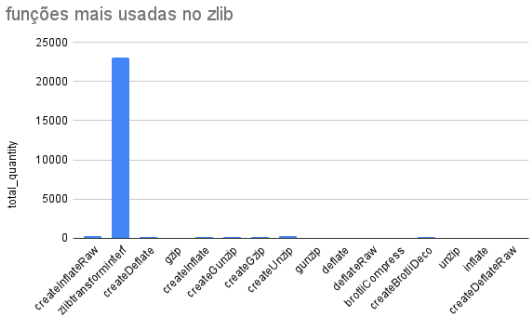
Fonte: Autoria própria

Figura 47: Funções assíncronas mais utilizadas do stream por projetos



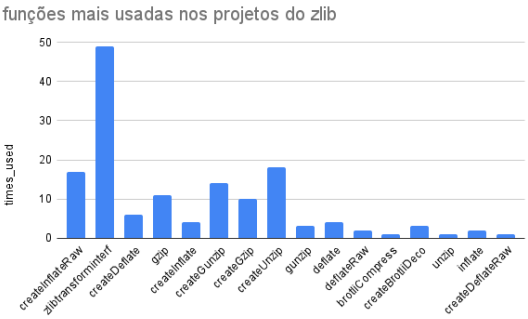
Fonte: Autoria própria

Figura 48: Funções assíncronas mais utilizadas do zlib



Fonte: Autoria própria

Figura 49: Funções assíncronas mais utilizadas do zlib por projetos



Fonte: Autoria própria

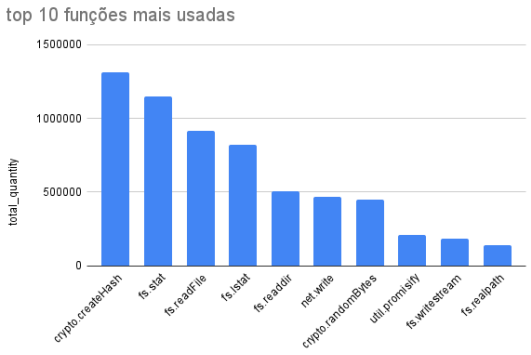
Ao analisar os gráficos gerados, é possível concluir que o módulo *fs* possui uma maior frequência de usos absolutos e por projetos entre todos os módulos (Figuras 28 e 29), contendo um uso absoluto de quatro funções com mais de 500 mil chamadas, além de cinco funções que estão em todos os projetos. Isto confirma o que foi analisado nas questões anteriores, sendo o *fs* o módulo mais popular nos casos de teste dos projetos

analisados. Outro módulo que tem um grande uso em projetos é o *child_process*, com duas funções sendo utilizadas em todos os projetos (Figuras 32 e 33).

Ainda analisando os gráficos, é possível perceber um certo padrão nos usos dos módulos de forma absoluta, no qual a maioria dos casos uma função é mais predominante usada do que o resto. Nos módulos *http, net, stream e zlib* (Figuras 38, 42, 46 e 48, respectivamente) este padrão é mais perceptível, no qual ocorre uma função sendo chamado mais vezes que as outras. Este é um comportamento esperado, pois a necessidade de uso de cada função são diferentes, no qual várias podem ser adequadas a casos mais específicos.

Para encontrar os padrões mais usados de forma geral, as Figura 50 e 51 representam as dez funções mais usadas de forma absoluta e em projetos, respectivamente. O eixo horizontal (eixo X) é representado pelo nome do módulo seguindo pela sua função.

Figura 50: Dez funções assíncronas mais utilizadas



Fonte: Autoria própria

Figura 51: Dez funções assíncronas mais utilizadas por projetos

Fonte: Autoria própria

Em relação ao uso absoluto (Figura 50), das cinco funções mais usadas, quatro são do módulo *fs*, reforçando a grande quantidade de utilizações analisadas anteriormente. Entretanto, a função mais utilizada é pertencente ao módulo *crypto, createHash*, é um resultado curioso pelo fato do módulo *fs* dominar as funções mais usadas, mas também demonstra a importância do módulo de criptografia nas aplicações. Um possível motivo para seu uso é na geração de identificadores únicos nos casos de teste, usando a função para gerar *hashes* em um ambiente controlado.

Analisando o uso por projetos (Figura 51), seis das dez mais usadas são do módulo *fs*, duas do módulo *crypto* e as duas restante do *child_process*. Entre essas, oito estão em todos os projetos analisados.

4.6 Considerações Finais

Após as análises apresentadas fornecem uma visão de como é a distribuição do uso dos módulos core em aplicações Node.js. Os módulos mais utilizados são o *fs* e *crypto*,

sendo dois módulos essenciais para aplicações reais. Outros que possuem um destaque é o *child_process* e *http*, ambos são importantes para o desenvolvimento e oferecem recursos importantes. Apesar do *http2* estar sendo desenvolvido para substituir o *http*, ainda existe pouca adesão do uso deste novo módulo, sendo preterido pela primeira até a data deste estudo.

Além disto, uma possível razão para a disparidade do módulo *fs* pode ser pelo uso da manipulação de arquivos para leitura, com o intuito de gerar dados para os testes.

Todas as conclusões observadas e retiradas das análises deste capítulo serão apresentadas no capítulo seguinte.

5 CONSIDERAÇÕES FINAIS

O objetivo do trabalho foi estudar projetos Node.js, buscando analisar o uso de funções assíncronas nos teste automatizados e assim responder às QPs propostas. Ao analisar os módulos core mais utilizados, foi perceptivo uma certa disparidade do *fs* em relação aos demais. Além disto, foi verificado que três módulos foram usados em todos os projetos (*fs, child_process e crypto*). Os outros módulos estão em menor quantidade, sendo o quarto mais usado (*http*) com um pouco mais de 100 projetos. Ainda em relação ao uso, sete módulos não foram usados, o motivo pode ser pelo módulo ser muito específico para um cenário.

Em relação aos padrões, era esperado uma certa preferência pelo *simple callback e return object* e foram confirmadas. Ambos os padrões foram muito utilizados, estando presente em ao menos um teste de cada projeto. O terceiro mais usado foi o *direct delay*, que chegou perto de quase duzentos projetos. Tanto o *object property como return promise* não foram, praticamente, utilizados.

Focando nas funções, foi analisado que as mais usadas pertenciam aos módulos de mais destaques. Ainda assim, mesmo com a disparidade do *fs*, uma função do *crypto* foi a mais utilizada em números brutos, apesar de não ser usada em todos os projetos. Apenas oito funções estiveram em todos os projetos, sendo distribuídos apenas em três módulos. Ainda importante ressaltar, como vários módulos tiveram uma função com um grande número bruto de usos e o resto com poucos. Por exemplo, a função *write* do módulo *net* esteve presente nas dez mais usadas, mas as outras funções tiveram um número bem inferior.

Por fim, é possível melhorias para trabalhos futuros. Possivelmente expandir a ferramenta criada, gerando uma forma de se adaptar a diferentes tipos de registros. Por exemplo, ser possível selecionar os dados que vão ser salvos e sendo recebido um *template* para se adaptar ao novos tipos de dados.

Referências

1 DAIGLE, K. *Octoverse: The state of open source and rise of AI in 2023*. 2023. Acessado em 18/01/2024. Disponível em: <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>.

2 NODE.JS. *Site do Node.js*. 2023. Acessado em 18/12/2023. Disponível em: <https://nodejs.org/en/about>.

3 ENDO, A. T.; MØLLER, A. NodeRacer: Event race detection for Node.js applications (to appear). In: *13th IEEE International Conference on Software Testing, Verification and Validation, ICST 2020, Porto, Portugal, March 23-27, 2020*. [S.l.]: IEEE Computer Society, 2020.

4 FOWLER, M. *In Memory Test Database*. 2005. Acessado em 18/01/2024. Disponível em: <https://martinfowler.com/bliki/InMemoryTestDatabase.html>.

5 AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

6 V8. *Site da Engine V8 da Google*. 2024. Acessado em 08/01/2024. Disponível em: <https://v8.dev>.

7 STATEOFJAVASCRIPT. *Library Tier List*. 2022. Acessado em 18/01/2024. Disponível em: <https://2022.stateofjs.com/en-US/libraries/>.

8 JEST. *Site do Jest*. 2023. Acessado em 11/12/2023. Disponível em: <https://jestjs.io/pt-BR/>.

9 MOCHA. *Site do Mocha*. 2023. Acessado em 11/12/2023. Disponível em: <https://mochajs.org/>.

10 GANJI, M.; ALIMADADI, S.; TIP, F. Code coverage criteria for asynchronous programs. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. (ESEC/FSE 2023), p. 1307–1319. ISBN 9798400703270. Disponível em: <https://doi.org/10.1145/3611643.3616292>.

11 WANG, J. et al. A comprehensive study on real world concurrency bugs in node.js. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 520–531.

12 ZHOU, J. et al. Nodert: Detecting races in node.js applications practically. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023. (ISSTA 2023), p. 1332–1344. ISBN 9798400702211. Disponível em: <https://doi.org/10.1145/3597926.3598139>.